Neural Network Parameter Estimation for the Modified Bistatic Scattering Strength Model (BISSM)

B. S. Bourgeois
Mapping, Charting and Geodesy Division
Ocean Science Directorate

0.1 Abstract

This technical note investigates the estimation of environmental parameters in the Bistatic Scattering Strength Model (BISSM) given backscatter strength and bathymetric data. A monostatic version of the model is derived, since this will be the form of data provided by acoustic imaging sensors. Feedforward neural networks, using the backpropagation learning algorithm, are used to perform the estimation of parameters for the nonlinear BISSM equation. The parameters that can be estimated are identified, and neural networks have been developed to estimate these parameters. Using noise-free artificial data generated with the BISSM equation, the networks provided excellent estimates of the desired parameters.

The primary impetus for this work is a need for the Naval Oceanographic Office (NAVOCEANO) to provide relevant survey support for Low-Frequency Active Acoustics (LFAA) programs and future Low-Frequency Active (LFA) operational systems. It has been recognized by the Commander, Naval Oceanographic Command (CNOC) that such support will require knowledge of certain bottom and subbottom properties and high-resolution geomorphology.

The BISSM model has been proposed as a model for aspects of active bottom reverberation. It was postulated that the parameters that activate the BISSM algorithm, might be measurable with NAVOCEANO's swath bathymetry system - SASS phase IV. This latest version of the SASS system developed by the Naval Air Development Center (NADC) generates 91 one-degree beams that can reach grazing angles down to 45 degrees and records backscattering strength as a function of grazing angle. Future SASS systems are expected to reach grazing angles down to 30 degrees. Although BISSM is designed as a LFAA prediction tool and should be most valuable as such below 1kHz, it should be scaleable to the higher frequency of 12kHz used by SASS and useful for inverse applications. The system parameters of SASS will determine the characteristics of the measurement process.

0.2 Acknowledgments

The author acknowledges the Commander of the Naval Oceanography Command (CNOC) for funding this project under program element O & M,N , managed by Mr. C. Wilcox. Special thanks are given to Dr. J.W. Caruthers of NOARL for his discussions regarding the BISSM equation, and to Mr. Mike Harris of NOARL for his interest and support of this project. The mention of commericial products or the use of company names does not in anyway imply endorsement by the U.S. Navy or NOARL.

Mr. Chester Wilcox, CNOC, has funded a small two-task effort, under the coordination of Dr. J.W. Caruthers, NOARL, to test the ability of BISSM to 1) predict basin reverberation at low-frequencies when given the necessary parameters and 2) measure parameters at high-frequency that are useful at the lower frequencies. This work addresses the second of the tasks and here we presume that BISSM is valid as a low-frequency predictor and as a high-frequency measurement tool of parameters needed for the low- frequency application and we test sensitivity of the measurement results to variations of the parameters.

Contents

0.1 Abstract i
0.2 Acknowledgments ii

1 Introduction **1**

2 Monostatic Reduction of BISSM **2**

3 Independence of BISSM Parameters **6**

4 Model Parameter Estimation from Monostatic Data **14**
4.1 Parameter Estimation Methods 14
4.2 Review of Neural Networks 15

5 Results **18**
5.1 Mackenzie Coefficient (μ) Estimation 19
5.2 m Estimation 20
5.3 δ Estimation 22

6 Summary **24**

A Neural Network Subroutine **26**

B Neural Network Training Program **32**

C BISSM Subroutine **44**

D Parameter Estimation Programs **47**
D.1 μ Parameter Estimation 47
D.2 nm Parameter Estimation 54
D.3 δ Parameter Estimation 61

Neural Network Parameter Estimation for the Modified Bistatic Scattering Strength Model (BISSM)

Chapter 1

Introduction

This paper investigates estimation of the environmental parameters in the Bistatic Scattering Strength Model (BISSM). The BISSM model has recently been proposed by Caruthers et al. [1] as an advancement of the model previously developed at the Naval Oceanographic and Atmospheric Research Laboratory (NOARL). The end goal is to use the backscatter and bathymetric data obtained from advanced acoustic imaging sensors to determine the parameters in the BISSM model. Thus, given backscatter strength and the incident and azimuth angles determined from bathymetry, it is desired to compute the ratio of sound speeds and densities at the bottom interface, the Mackenzie coefficient, the root mean square (RMS) microscale heights roughness, and the fine-scale RMS slopes in the along track and across track directions.

The approach taken in this technical note is to use artificial data, generated by the BISSM equation, to estimate the known parameters that were used to generate the data. Since BISSM is a nonlinear equation it cannot be inverted directly, requiring an error minimization approach for the estimation of its parameters. Using artificial data provides an opportunity to determine the best possible performance of the estimation techniques. Neural networks are used in this work to estimate the desired parameters, given an input data vector of backscatter strength as a function of incident angle. Essentially, this work shows proof of concept, using noise free data.

In the next chapter, the BISSM equation is presented, and the monostatic version of this equation is derived. A monostatic version is used in this work since this is the form of data that will be collected by acoustic imaging systems. Chapter 3 investigates the sensitivity of the backscatter strength produced by the BISSM equation to its parameters. By examining the correlation between the effects of the various parameters, a set of parameters that have potential for estimation are determined. In Chapter 4 various multidimensional, nonlinear minimization approaches are discussed, and the advantage of a neural network approach over more traditional approaches is explained. A quick review of feedforward neural networks and the backpropagation learning algorithm is also given in this chapter. Chapter 5 presents the results of the parameter estimation attempts, showing that the trained neural networks reliably provide good estimates. Finally, Chapter 6 summarizes the significant results, and details further research that is required.

Chapter 2

Monostatic Reduction of BISSM

As stated in the previous chapter, the goal of this task is to estimate the environmental parameters in the BISSM equation given collocated bathymetric and backscatter data. Specifically, it is desired to estimate n, the ratio of sound speeds, m, the ratio of densities, μ, the Mackenzie coefficient, σ, the RMS microscale heights roughness, and δ_x and δ_y, the fine-scale RMS slopes in the x and y directions. Potential sources of data for this inversion are multibeam hull mounted sonar and towed sidescan sonar systems. In both cases the source and receiver will be at the same position so that $\theta = \theta_i = \theta_s$ and $\phi = \phi_i = \phi_s - \pi$, where θ_i is the source incident angle, θ_s is the receiver scattered angle, and ϕ_i and ϕ_s are the incident and scattered azimuth angles. Thus, the first step in parameter estimation is to reduce the bistatic model to a monostatic model.

The BISSM bistatic equation [1] is given by:

$$m_{bs} = m_1 + m_2 \, . \tag{2.1}$$

In (2.1) m_1 is the incoherent term and is given by:

$$m_1 = \mu \sin\theta_i \sin\theta_s \, . \tag{2.2}$$

Both angles are measured upward from the local bottom facet plane. In (2.2) μ is the Mackenzie coefficient.

In (2.1) m_2 is the coherent term and is given by:

$$m_2 = R_0^2 \exp(-g) \frac{F^2}{2\pi\delta_x\delta_y} \exp\left\{-\frac{1}{2q^2}\left(\frac{X_x^2}{\delta_x^2} + \frac{X_y^2}{\delta_y^2}\right)\right\} \, . \tag{2.3}$$

R_0, in (2.3), is the Rayleigh reflection coefficient between two fluids which is given by:

$$R_0 = \frac{m\sin\theta_i - \sqrt{n^2 - \cos^2\theta_i}}{m\sin\theta_i + \sqrt{n^2 - \cos^2\theta_i}} \tag{2.4}$$

where m is the ratio of densities and n is the ratio of sound speeds. In (2.3) g is the square of the Rayleigh roughness parameter given by:

$$g = \sigma^2 q^2 \tag{2.5}$$

where σ is the RMS microscale heights roughness. q is given by:

$$q = k\left(\sin\theta_i + \sin\theta_s\right) \tag{2.6}$$

and k is the acoustic wavenumber given by $2\pi/\lambda$, where λ is the wavelength of the acoustic source. F is given by:

$$F = \frac{1}{2}\left(1 + \frac{X^2}{q^2}\right) \tag{2.7}$$

where

$$X^2 = X_x^2 + X_y^2$$

and

$$\begin{aligned} X_x &= k\left(\cos\theta_s\cos\phi_s - \cos\theta_i\cos\phi_i\right) \\ X_y &= k\left(\cos\theta_s\sin\phi_s - \cos\theta_i\sin\phi_i\right) . \end{aligned} \tag{2.8}$$

ϕ_i and ϕ_s are measured clockwise from North to the projection of the vector onto the local horizontal plane. Finally, δ_x and δ_y are the fine-scale RMS slopes in the x and y directions, and are given by:

$$\delta_x^2 = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\left(\delta_x^{ij}\right)^2 \tag{2.9}$$

$$\delta_y^2 = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\left(\delta_y^{ij}\right)^2 \tag{2.10}$$

where n is the number of pixels along one side of a square grid, and δ_x^{ij} and δ_y^{ij} are the incremental change in slope in the x and y directions. The equation's parameters are summarized in Table 2.1.

For the monostatic case, the incoherent term (2.2) simplifies to:

$$\begin{aligned} m_1 &= \mu\sin\theta\sin\theta \\ &= \mu\sin^2\theta . \end{aligned} \tag{2.11}$$

Also, X_x becomes

$$\begin{aligned} X_x &= k\left[\cos\theta\left(\cos\phi - \cos\left(\phi - \pi\right)\right)\right] \\ &= k\left[\cos\theta\left(\cos\phi + \cos\phi\right)\right] \\ &= 2k\left(\cos\theta\cos\phi\right) \end{aligned}$$

and X_y becomes

$$\begin{aligned} X_y &= k\left[\cos\theta\left(\sin\phi - \sin\left(\phi - \pi\right)\right)\right] \\ &= k\left[\cos\theta\left(\sin\phi + \sin\phi\right)\right] \\ &= 2k\left(\cos\theta\sin\phi\right) . \end{aligned}$$

Table 2.1: Parameters in the BISSM model.

Parameter	Description	Units
n	ratio of sound speeds	unitless
m	ratio of densities	unitless
μ (mu)	Mackenzie coefficient	unitless
σ (sigma)	RMS micro-scale heights roughness	meters
δ_x (delx)	fine-scale RMS slopes in x direction	radians
δ_y (dely)	fine-scale RMS slopes in y direction	radians
θ_i	source incident angle	radians
θ_s	receiver scattered angle	radians
θ	monostatic incident angle	radians
ϕ_i	source azimuth angle	radians
ϕ_s	receiver azimuth angle	radians
k	wavenumber	radians/meter

q reduces to

$$q = 2k \sin\theta$$

and from (2.3) we have:

$$\begin{aligned} & \exp\left\{-\frac{1}{2q^2}\left(\frac{X_x^2}{\delta_x^2}+\frac{X_y^2}{\delta_y^2}\right)\right\} \\ = \; & \exp\left\{-\frac{1}{8k^2\sin^2\theta}\left(\frac{4k^2\cos^2\theta\cos^2\phi}{\delta_x^2}+\frac{4k^2\cos^2\theta\sin^2\phi}{\delta_y^2}\right)\right\} \\ = \; & \exp\left\{-\frac{\cot^2\theta}{2}\left(\frac{\cos^2\phi}{\delta_x^2}+\frac{\sin^2\phi}{\delta_y^2}\right)\right\} . \end{aligned} \tag{2.12}$$

Finally, F can be simplified to

$$\begin{aligned} F &= \frac{1}{2}\left(1+\frac{X^2}{q^2}\right) \\ &= \frac{1}{2}\left(1+\frac{4k^2\cos^2\theta\cos^2\phi+4k^2\cos^2\theta\sin^2\phi}{4k^2\sin^2\theta}\right) \\ &= \frac{1}{2}\left(1+\frac{4k^2\cos^2\theta\left(\cos^2\phi+\sin^2\phi\right)}{4k^2\sin^2\theta}\right) \\ &= \frac{1}{2}\left(1+\frac{\cos^2\theta}{\sin^2\theta}\right) \\ &= \frac{1}{2}\csc^2\theta \end{aligned}$$

and

$$F^2 = \frac{1}{4\sin^4\theta} .$$

Thus, (2.1) becomes

$$m_{ms} = \mu \sin^2\theta + \frac{R_0^2 \exp(-g)}{8\pi\delta_x\delta_y \sin^4\theta} \exp\left\{-\frac{\cot^2\theta}{2}\left[\frac{\cos^2\phi}{\delta_x^2} + \frac{\sin^2\phi}{\delta_y^2}\right]\right\} . \tag{2.13}$$

Note that if we assume $\delta = \delta_x = \delta_y$ then (2.12) simplifies to

$$\exp\left\{-\frac{\cot^2\theta}{2\delta^2}\right\}$$

and (2.13) simplifies to

$$m_{ms} = \mu \sin^2\theta + \frac{R_0^2 \exp(-g)}{8\pi\delta^2 \sin^4\theta} \exp\left\{-\frac{\cot^2\theta}{2\delta^2}\right\} . \tag{2.14}$$

Given collocated bathymetric and backscatter data for a region, we desire to determine the values of n, m, μ, σ, δ_x and δ_y that provide the least mean square error between the actual data (for a single scan line) and (2.13). The angles θ and ϕ are determined from the bathymetric data and the source ray trace angle (with respect to the local horizontal plane), and k is determined from the source frequency and the local speed of sound in the water (assumed constant).

Chapter 3

Independence of BISSM Parameters

As previously stated, it is desired to estimate the parameters n, m, μ, σ, δ_x and δ_y in (2.13) given collocated bathymetric and backscatter data. Successful estimation of these parameters requires that their effect on the computed backscatter intensity be uncorrelated with each other. In this chapter the correlation between the desired parameters is determined empirically in order to identify those parameters that may potentially be estimated. The ranges of interest for the desired parameters for abyssal plains are given in Table 3.1. A wavenumber of 5.0265 radians/meter is used, which corresponds to a sound velocity of 1500 m/sec and a sonar frequency of 1.2 kHz.

Table 3.1: Parameter ranges for the BISSM model

Parameter	Nominal	Low	High
n (unitless)	0.99	0.97	1.2
m (unitless)	1.4	1.2	1.8
μ (unitless)	0.002	0.0002	0.02
σ (meters)	0.01	0.005	0.02
δ_x (radians)	0.05236	0.01745	0.08727
δ_y (radians)	0.05236	0.01745	0.08727

Figure 3.1 shows the backscatter strength versus incident angle with all six parameters at their low, nominal, and high values. In this figure the angle ϕ has been set to zero. Figure 3.1 illustrates three dominant motions in the backscatter strength curve as the parameter values are varied. One motion is a fairly uniform rise in backscatter intensity at lower incident angles (less than 70 degrees) as the parameters are increased. Secondly, the 'pivot point', where the backscatter intensity curves sharply upward, moves to lower incident angles as the parameter values are increased. Thirdly, the slope of the sharp rise at higher angles decreases with an increase in parameter values.

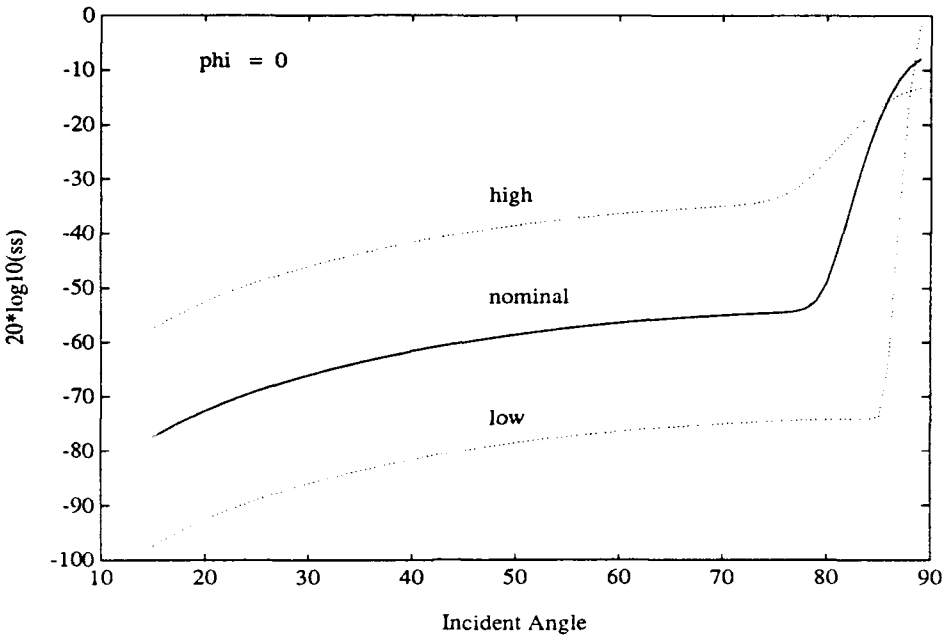

Figure 3.1: Scattering Strength Curves for all parameters at their high, nominal, and low values.

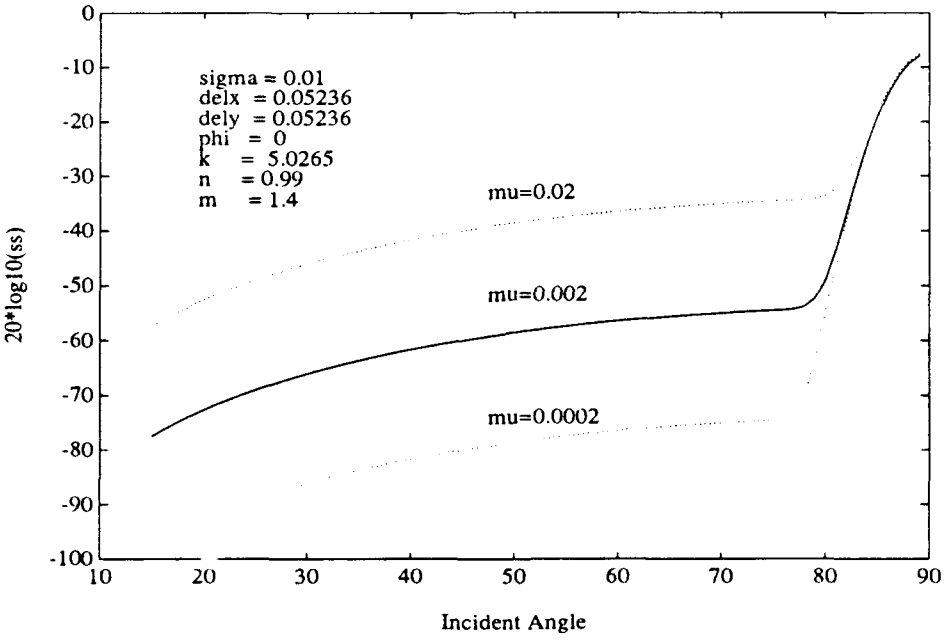

Figure 3.2: Scattering Strength Curves for μ at high, nominal, and low values, with all other parameters nominal.

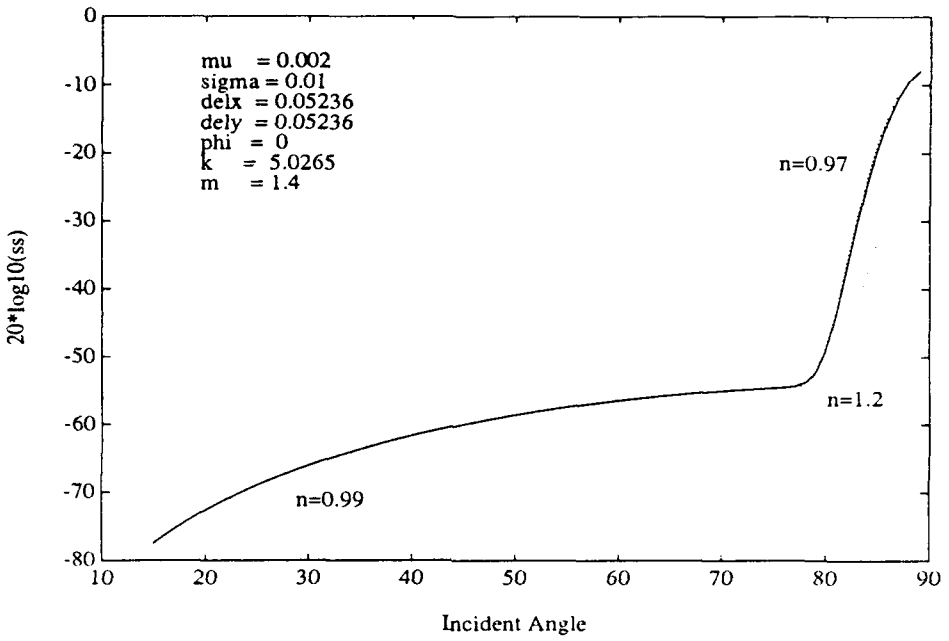

Figure 3.3: Scattering Strength Curves for n at high, nominal, and low values, with all other parameters nominal.

Figure 3.2 shows the effect on backscatter strength due to variations in μ with all other parameters at their nominal values. It is seen in this figure that an increase in μ will result in a fairly uniform increase of the backscatter strength for lower incident angles (less than 70 degrees). From the remaining figures shown in this chapter, it is seen that μ is the dominant parameter for backscatter intensity changes at low incident angles.

Figures 3.3 through 3.5 show the effect on backscatter strength due to variations in the n and m parameters. In Figure 3.3 n is varied and all other parameters have nominal values, in Figure 3.4 m is varied, and in Figure 3.5 n and m are varied simultaneously. Both parameters cause a change in the slope of the sharp rise at high incident angles, where an increase in n causes a decrease in slope, and an increase in m causes an increase in slope. It is seen from these figures that the effect of changes in n and m on the backscatter strength are highly correlated, so n and m cannot be independently estimated. However, when n is large m is also large, so a simplifying assumption will be used during parameter estimation that the two parameters are linearly related. This essentially reduces the parameter estimation to a single parameter, m or n, and m will be estimated in this work. Figure 3.6 shows that the effect on backscatter strength due to changes in σ are highly correlated with those due to changes in n and m. Furthermore, the variations due to changes in σ are extremely small. Consequently, σ cannot be successfully estimated for a frequency of 1.2 kHz, especially since actual data will be contaminated by noise.

Figures 3.7 through 3.11 show the effect on backscatter strength due to variations in δ_x, δ_y, and ϕ. In Figures 3.7 and 3.8 ϕ is zero, and it is seen that δ_x shifts the pivot

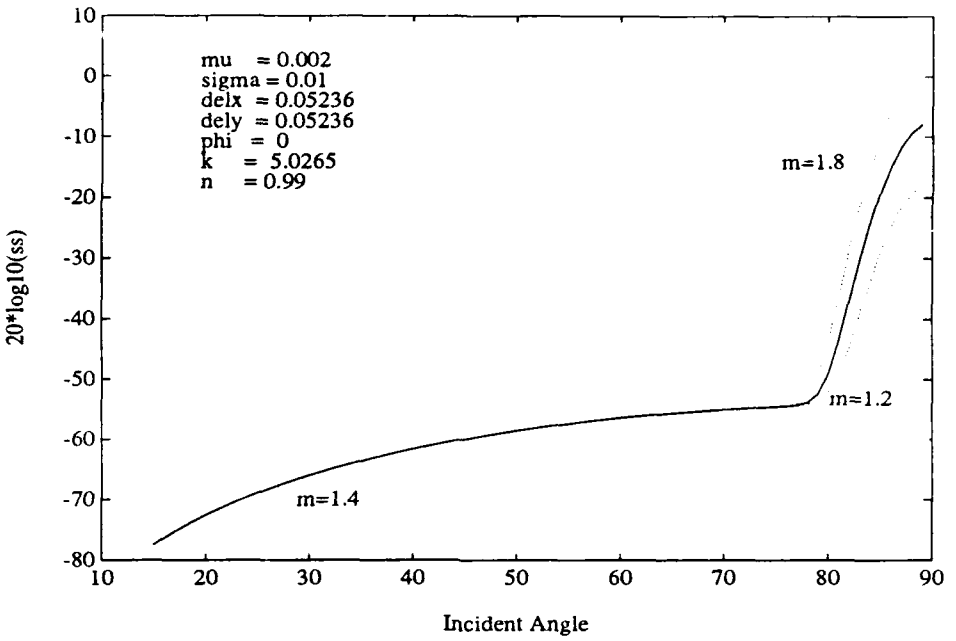

Figure 3.4: Scattering Strength Curves for m at high, nominal, and low values, with all other parameters nominal.

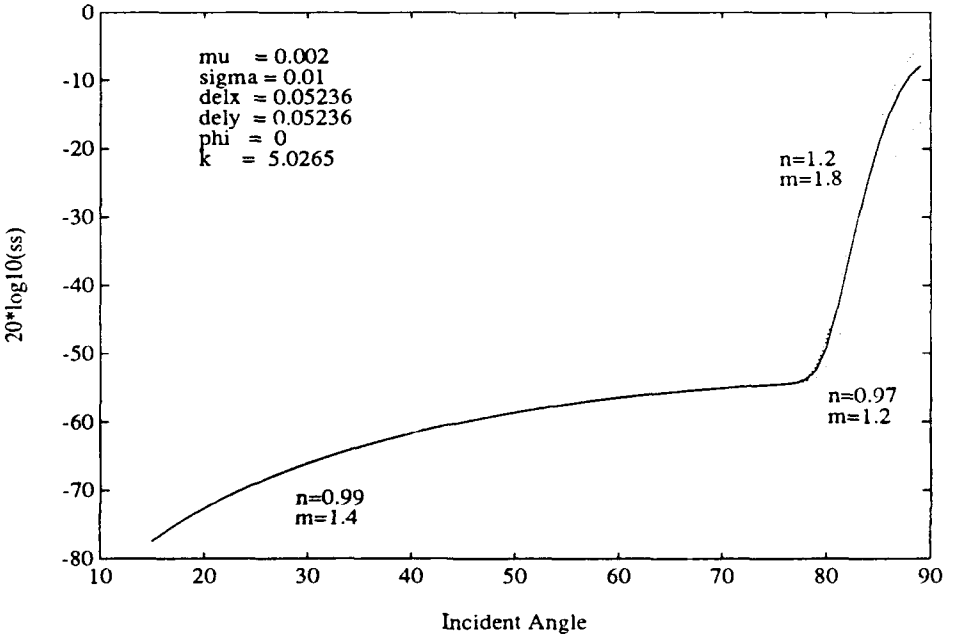

Figure 3.5: Scattering Strength Curves for both n and m at high, nominal, and low values, with all other parameters nominal.

point and affects the slope at high incident angles, while δ_y only affects the slope. This result is expected from the form of (2.13) since $\sin\phi$ is zero when ϕ is zero. In Figures 3.9 and 3.10 ϕ is 45 degrees, so the effect on backscatter strength due to changes in δ_x and δ_y are equal. These figures show that the effect of changes in δ_x and δ_y are highly correlated, so that these parameters cannot be independently estimated. To simplify the problem, it will be assumed that $\delta = \delta_x = \delta_y$, and this single parameter will be estimated. As shown in (2.14), when we set $\delta_x = \delta_y$ the parameter ϕ vanishes. In Figure 3.11 δ_x and δ_y are varied simultaneously. The effect on backscatter strength due to the δ parameter is seen to be a shift in the incident angle of the pivot point, and a change in the slope at higher angles. While there is some correlation between the δ and the m parameter as seen by the change in high angle slope, the change in pivot point due to δ may provide enough unique information to distinguish between changes in backscatter strength due to δ versus nm.

Summarizing, it has been observed that only three parameters may potentially be estimated from actual backscatter data using the BISSM (monostatic version) relationship: μ, m, and δ. The changes in the μ parameter result in a shift up or down of the backscatter strength at low angles of incidence (less than 70 degrees). The effect of the n and m parameters on backscatter strength were seen to be highly correlated, so only one of these parameters can be estimated. A linear relationship will be assumed between n and m to perform the estimation, so only m will be estimated. The primary effect of the n and m parameters is to change the slope of the curve at high incident angles. It was observed that the effect of σ on backscatter strength was too small to be reliably estimated at the frequency used in this study. Also, the effects of σ and nm are highly correlated. The effects of δ_x and δ_y were seen to be highly correlated, depending upon the value of ϕ. Consequently, these terms are combined into a single parameter δ. The δ parameter affects the slope of the backscatter strength curve at high incident angles, which is correlated with the nm parameter, but it also shifts the pivot point in the curve transition from low to high incident angles.

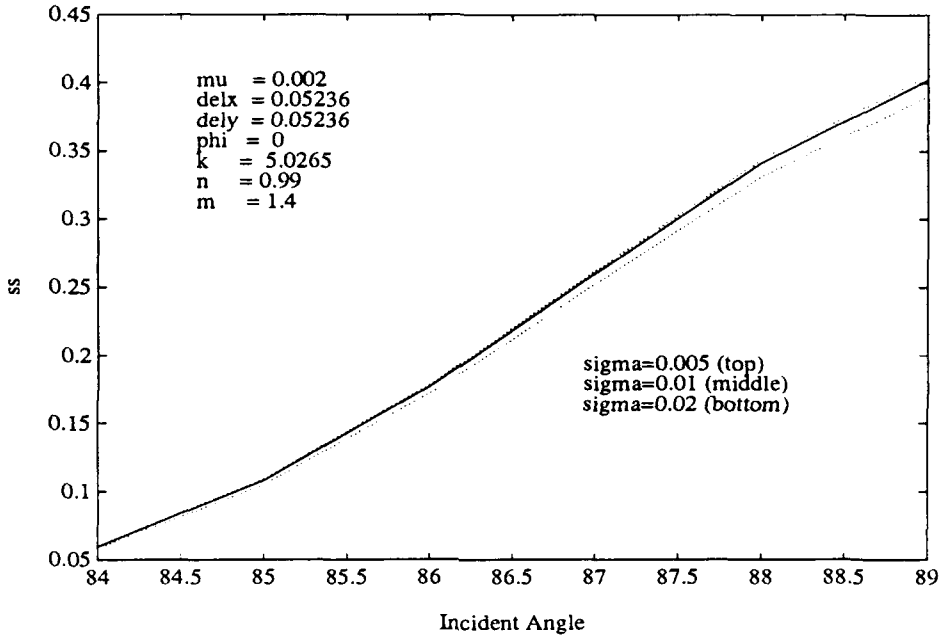

Figure 3.6: Scattering Strength Curves for σ at high, nominal, and low values, with all other parameters nominal.

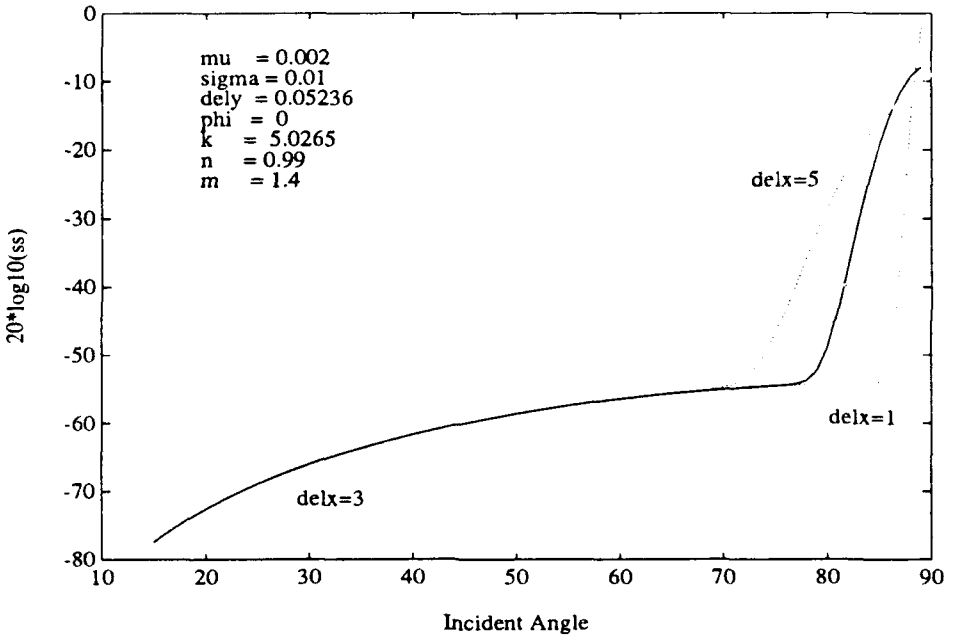

Figure 3.7: Scattering Strength Curves for δ_x at high, nominal, and low values, with all other parameters nominal, and $\phi = 0$ degrees.

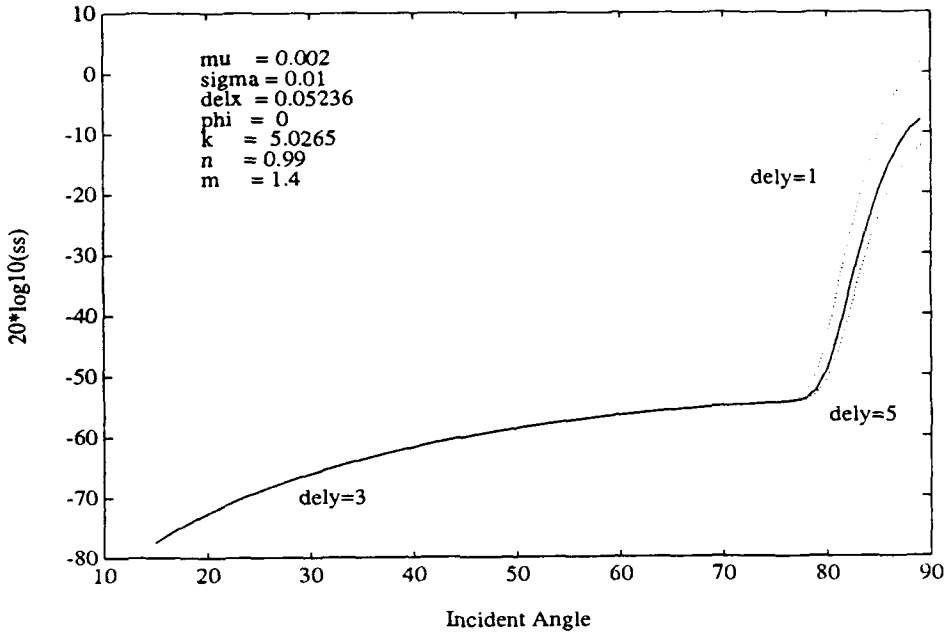

Figure 3.8: Scattering Strength Curves for δ_y at high, nominal, and low values, with all other parameters nominal, and $\phi = 0$ degrees.

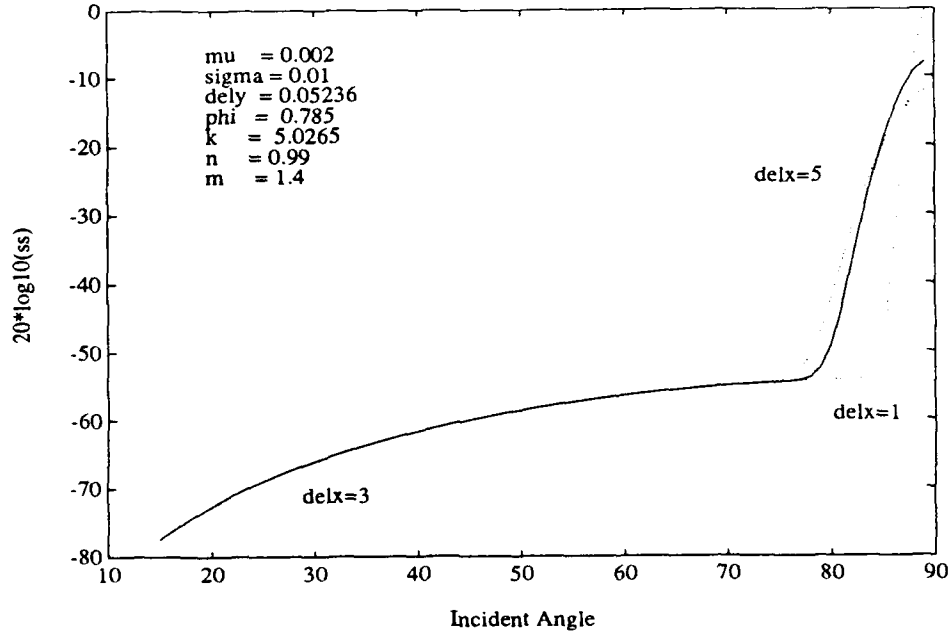

Figure 3.9: Scattering Strength Curves for δ_x at high, nominal, and low values, with all other parameters nominal, and $\phi = 45$ degrees.

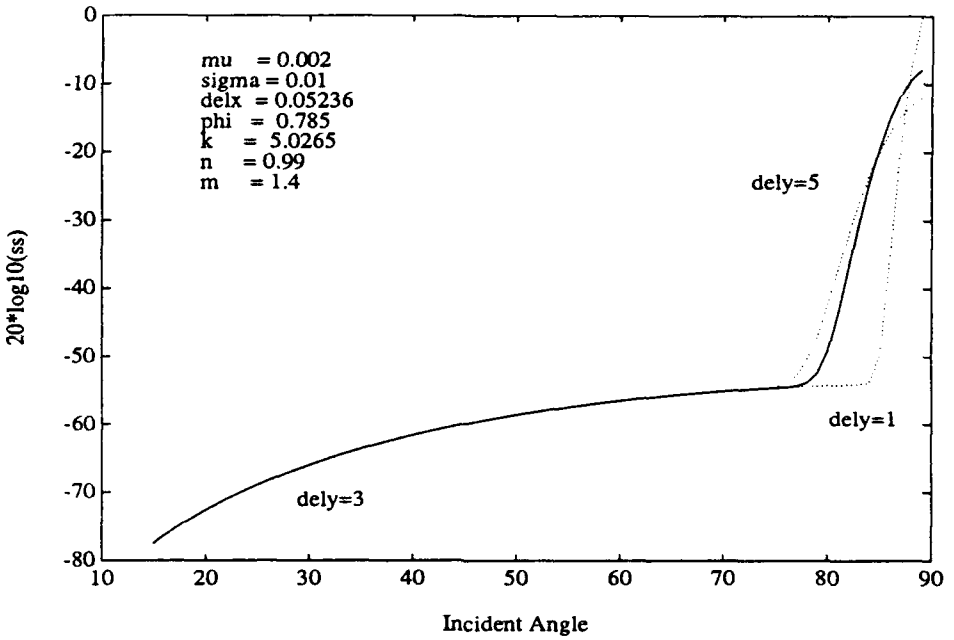

Figure 3.10: Scattering Strength Curves for δ_y at high, nominal, and low values, with all other parameters nominal, and $\phi = 45$ degrees.

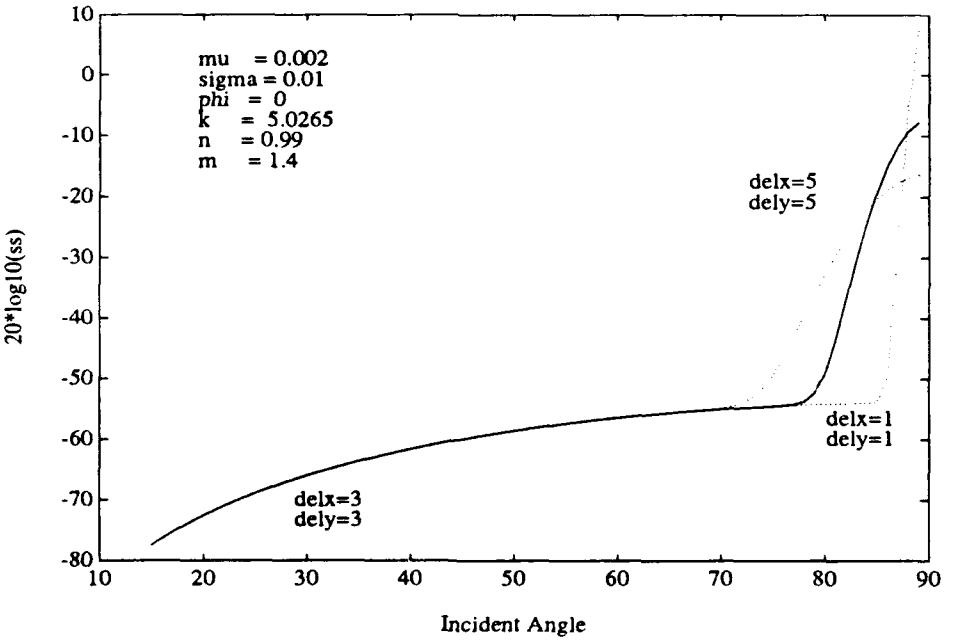

Figure 3.11: Scattering Strength Curves for both δ_x and δ_y at high, nominal, and low values, with all other parameters nominal, and $\phi = 0$ degrees.

Chapter 4

Model Parameter Estimation from Monostatic Data

4.1 Parameter Estimation Methods

As discussed in Chapter 3, it is desired to estimate the parameters μ, nm, and δ given backscatter strength, m_{bs}, incident angle, θ, and acoustic wavenumber, k. The data that will be used for the estimation is $m_{bs}(\theta)$, backscatter strength as a function of incident angle, and k will be assumed constant for this vector. The vector $m_{bs}(\theta)$ will typically be obtained from the backscatter strength and bathymetric data produced by the sonar systems mentioned earlier. A single vector, and corresponding wavenumber, is obtained for each transmit/receive cycle of these sonar systems. The work discussed in this paper demonstrates the ability to estimate the desired parameters by generating artificial backscatter data using the BISSM equation.

The estimation of μ, nm, and δ can be viewed as a minimization problem. We have $m_{bs}(\theta_i)$, the scattering strength at angle θ_i for $i = 1$ to n. We also have a function T, $\{T : T(\theta_i, \gamma)\}$, which is the BISSM equation and is assumed to be a model for m_{bs}. In T, γ is a vector composed of the model parameters: $\gamma = [m, \mu, \delta]$. Thus given θ_i and an estimate of γ we can compute an estimate for scattering strength, $m_{bs}(\theta_i)_e$. The mean square error between the estimate and the actual scattering strength is a function of γ, and is denoted $E(\gamma)$. We want to minimize this error, which is given by:

$$E(\gamma) = \frac{1}{n}\sum_{i=1}^{n}[m_{bs}(\theta_i) - T(\theta_i, \gamma)]^2 \tag{4.1}$$

where n is the total number of incident angles. By minimizing $E(\gamma)$, our estimate of γ approaches the actual value γ_0.

Thus, given initial estimates of these parameters, the BISSM equation is used to generate an estimate of the backscatter strength $m_{bs}(\theta)_e$. A better estimate of the parameters is obtained by iteratively adjusting them to minimize the mean square error between the actual backscatter, $m_{bs}(\theta)$, and the estimated backscatter $m_{bs}(\theta)_e$. Parameter adjustment is typically accomplished through random search or gradient techniques. While random search techniques can guarantee a global minimum error,

they are very computationaly expensive. There are several gradient search methods available for nonlinear, multidimensional minimization problems. Techniques that don't require computation of first derivatives include the downhill simplex method, due to Nelder and Mead [2], and Powell's method [3]. Methods that require the computation of first derivatives include the Polak-Ribiere algorithm [4] and the Broyden-Fletcher-Goldfarb-Shanno algorithm [5]. Gradient techniques can provide a global minimum if the error surface is relatively smooth. For highly irregular error surfaces, which are encountered with noisy data, it may be required to start the estimation with multiple initial conditions to ensure that the final solution is not a local minimum.

Gradient approaches are also fairly computationally intensive, and a gradient search must be performed for each new data vector $m_{bs}(\theta)$. Another approach for the estimation of γ is to use a feedforward neural network to model the inverse relationship, F, between the model T, and γ where

$$F(T(\theta, \gamma)) = \gamma \tag{4.2}$$

Assuming that T is a good model for m_{bs} then $F(m_{bs})$ will provide a good estimate of γ. As shown by Kolmorogov [6], a 2 hidden layer feedforward neural network can approximate any nonlinear R^m to R^n mapping function arbitrarily close, depending on the number of nodes in the hidden layers. Furthermore, neural networks are robust in the presence of noise. A significant advantage of a neural network approach over gradient search is that once the network is trained, it will directly produce an estimate of γ, without having to perform a gradient search for each new m_{bs} vector. Also, neural networks are well suited for implementation on parallel machines. The following section provides a brief review of feedforward neural networks and the backpropagation algorithm, which will be used for the estimation of γ in this work.

4.2 Review of Neural Networks

A neural network is a device that can be used to recognize signal phenomena or perform numeric functions. A neural network is composed of one or more layers, each containing one or more neurons. Each neuron in a feedforward type network typically accepts multiple inputs, where these inputs are signals provided to the network or are the outputs of neurons in a previous layer. A single neuron is essentially a linear combiner; it produces a weighted sum of it inputs. Additionally, each neuron's output is limited by a signum or sigmoid function. Signum functions, or hard limiters, are typically used for applications where the network is intended to make discrete classification decisions based on the inputs. Sigmoid functions, or soft limiters, are used for networks designed to perform various real valued signal processing tasks.

A neural network typically must be trained to perform a particular function. Training involves providing the network with a series of input and desired signals, and adapting the weights of the neurons within the network to minimize the error between the network's output and the desired signal. One of the more popular training algorithms is backpropagation [7], and this training method is reviewed here.

Updating network weights with the backpropagation algorithm is analogous to the process used by the adaptive least mean square (LMS) filter. The algorithm computes an estimate of the gradient of the mean square error with respect to the system's weights, and this information is used to move the weights so that the error approaches a minimum. It is important to note that a neural network is a nonlinear device, typically with many local minima. The existence of local minima often make it difficult for the network to reach its global minimum with unsupervised training. A derivation of the backpropagation algorithm is now presented.

Given $\mathbf{x}_k$ as the input vector at time k, and $\mathbf{w}_k$ as the weight vector at time k, the output of the linear summation of a single neuron is given by:

$$s_k = \mathbf{x}_k^T \mathbf{w}_k \tag{4.3}$$

and the output of the neuron is:

$$y_k = \mathrm{sgm}(s_k) \tag{4.4}$$

where sgm denotes a sigmoid function. The hyperbolic tangent is the sigmoid function used in this paper. The error here is defined as $e_k = d_k - y_k$ where d_k is the desired signal at time k.

The gradient of the mean square error with respect to the neuron's weights is given by:

$$\frac{\partial E(e_k^2)}{\partial \mathbf{w}_k} \tag{4.5}$$

where E denotes the expected value. By eliminating the expected value we obtain a stochastic estimate of the gradient:

$$\frac{\partial e_k^2}{\partial \mathbf{w}_k} \tag{4.6}$$

resulting in the following weight update equation:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mu \frac{\partial e_k^2}{\partial \mathbf{w}_k} \tag{4.7}$$

where μ is the learning gain that controls the speed of convergence. Taking the partial derivative we obtain the following:

$$\begin{aligned} \frac{\partial e_k^2}{\partial \mathbf{w}_k} &= 2e_k \frac{\partial e_k}{\partial \mathbf{w}_k} \\ &= 2e_k \frac{\partial (d_k - \mathrm{sgm}(s_k))}{\partial \mathbf{w}_k} \\ &= -2e_k \mathrm{sgm}'(s_k) \frac{\partial s_k}{\partial \mathbf{w}_k} \\ &= -2e_k \mathrm{sgm}'(s_k) \mathbf{x}_k \end{aligned} \tag{4.8}$$

However, the sigmoid function used here is tanh, so we have

$$\begin{aligned} \mathrm{sgm}'(s_k) &= \frac{d \tanh(s_k)}{d s_k} \\ &= 1 - \tanh^2(s_k) \\ &= 1 - y_k^2 \end{aligned} \tag{4.9}$$

and thus the weight update equation for a single neuron becomes:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + 2\mu e_k \left(1 - y_k^2\right) \mathbf{x}_k \tag{4.10}$$

In a multilayer neural network, desired signals are typically available only for the neurons in the output layers. A general weight update equation for any neuron is given by Widrow and Lehr [7] as:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + 2\mu\delta_k \mathbf{x}_k \tag{4.11}$$

where δ_k is called the square error derivative for the particular neuron, and $\mathbf{x}_k$ is the input vector for that neuron. For the output layer neurons, where desired signals are available, δ_k in (4.11) is obtained from (4.10) and is given by $\delta_k = e_k(1 - y_k^2)$. For a neuron not in the output layer δ_k is given by [7]:

$$\delta_{m,k}^l = e_{m,k}^l \mathrm{sgm}'(s_{m,k}^l) \tag{4.12}$$

where $\delta_{m,k}^l$ is the square error derivative at time k for the m^{th} neuron in the l^{th} layer, $s_{m,k}^l$ is the output of the same neuron (before the sgm function) at time k, and $e_{m,k}^l$ is the backpropagated error for that neuron at time k. The backpropagated error is given by:

$$e_{m,k}^l = \sum_{i=1}^{q} \delta_{i,k}^{l+1} w_{(m,l)\to i,k}^{l+1} \tag{4.13}$$

where $w_{(m,l)\to i,k}^{l+1}$ is the weight that connects the m^{th} neuron in the l^{th} layer to the i^{th} neuron in the $(l+1)^{st}$ layer. Thus the backpropagated error for a neuron that is not in the output layer is given by the sum of the square error derivatives of the neurons in the following layer, each scaled by the neuron weight that connects the neuron being evaluated with the neuron in the next layer.

Chapter 5

Results

In this chapter the results for estimation of each of the three parameters in γ are shown, using noise-free artificially generated data. To perform the estimation a neural network program was written in 'C' using the backpropagation algorithm [7]. This program is given in Appendix A, and the program that controls it during training is in Appendix B. The program implements a two hidden layer network with a single output neuron, and provides the capability to select the number of neurons in the hidden layers, and the number of inputs to the first hidden layer. The number of neurons in each of the two hidden layers is identical, and full connectivity between layers has been used thus far. A weight jogging capability has also been included, which allows the addition of small amounts of random noise to the weights to 'jog' the network out of local minima. This feature is paramount since, as illustrated by Widrow and Lehr [7], the error surface of a neural network is typically rich in local minima. The computation time has been found to be a nearly linear function of the total number of weights in the network. With learning on the computation time is about $4.56 \cdot 10^{-4}$ seconds/iteration·tap on an AT computer. With learning off the time is about one third of this value, thus the backpropagation process requires about two thirds of the total computation time. Tests on a Sun 4 computer yielded a computation time of $2.13 \cdot 10^{-5}$ seconds/iteration·tap, approximately 21 times as fast as the AT.

The network architecture that is best for a given problem must be determined empirically, typically through iterative training and testing of different architectures. The method used here is to start with a small network, and to increase its size to the point where a significant reduction in error is no longer obtained. The data sets for training are generated by randomly varying the parameters in the BISSM equation, providing very large training sets. The BISSM equation is implemented in the program given in Appendix C. The training method used here starts with a large learning gain for fast convergence, followed by successive reductions in gain to achieve lower error. Further estimation improvement may be possible by adding more layers, or through advanced techniques such as momentum, feedback, or feedforward from nonadjacent layers.

The following sections discuss the architecture, training, and results for estimation of μ, m, and δ. With the tanh function a neuron's output is limited to the range of

$[-1, 1]$, so the data provided to the network is scaled and in some cases normalized to improve the network's estimation. The range of values for μ and δ are sufficiently small and do not require scaling. The values for n and m may be large, so the network is trained to estimate one-fourth of their value. As previously discussed, a linear relationship is assumed between n and m. The relationship used here is $n = m * .3833 + .51$, allowing for the full range of both parameters. Initial training attempts held two of the parameters constant and randomly varied the parameter of interest. It was found that the error could be made small for the parameter being estimated, but could become large if the other two parameters were varied from the values used during training. This is a consequence of some correlation existing between the three parameters. To reduce this problem, all three parameters are varied randomly throughout their full ranges during training.

5.1 Mackenzie Coefficient (μ) Estimation

Table 5.1: μ network training

Learning Gain	Iterations	Average Percent Error
0.8	500	136.4
0.8	500	84.6
0.6	600	42.5
0.6	600	20.3
0.4	800	7.75
0.4	800	4.16
0.2	1000	1.89
0.2	1000	1.30
0.1	800	0.87
0.1	800	0.72
0.05	800	0.58
0.05	800	0.52
0.01	800	0.50
0	800	0.47

For the estimation of μ the input vector given to the network is the backscatter strength for incident angles ranging from 15 to 60 degrees in 2 degree increments, where each input to the network corresponds to a specific incident angle. Angles higher than 60 degrees are not required since the dominant effect of μ is at lower angles. Angles below 15 degrees cannot be used since the incident angle must be large enough with respect to n in the Rayleigh reflection coefficient calculation to avoid a complex result. An angle increment of 2 degrees was found to be sufficient to obtain small estimation errors. The error could possibly be reduced by using a smaller increment, but this will also require a larger network. Since the backscatter

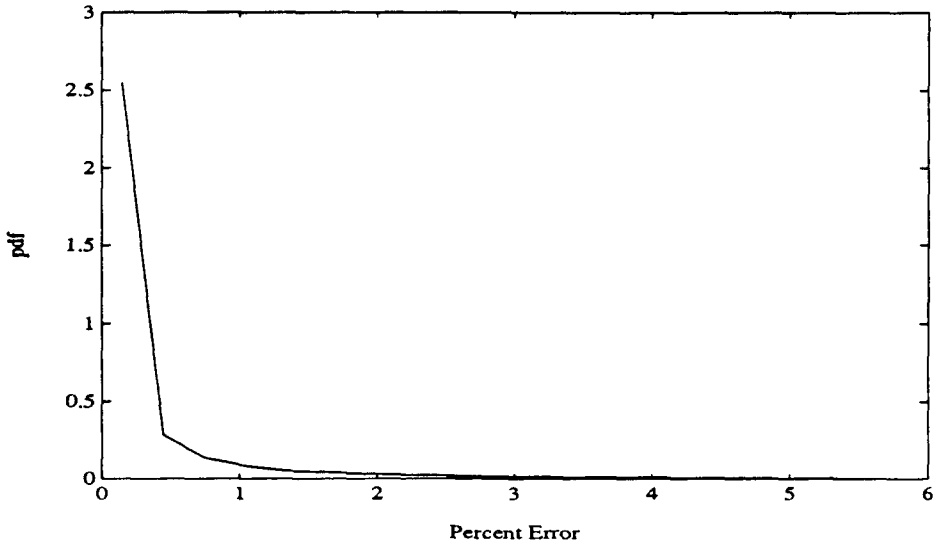

Figure 5.1: Probability density function of the percent estimation error for the μ estimating program. The scattering strength data was obtained by randomly varying δ, μ, and m.

strength is small at lower angles (ranging from about 10^{-6} to 10^{-2}), the data vector is multiplied by 10 before giving it to the network. Also, the data vector was 'centered' by subtracting 0.15 from all values. This allows the data at the lower angles to have approximately equal influence on the training process as the higher angles.

The final configuration for the μ network has 23 inputs to each of the nodes in the first layer and 10 nodes in each of the two hidden layers. The μ estimation program is given in Appendix D.1. The final training sequence is shown in Table 5.1, and an average percent error of 0.47 was obtained with training off (gain = 0). As shown in this table the network converged very rapidly to a small error. Faster computation rates can be obtained by increasing the angle increment in the input data vector, probably with a minimal increase in estimation error. The probability density function of the percent estimation error is shown in Figure 5.1, and indicates that for the majority of the test data the estimation error is below 1 percent. As expected, the estimation of μ was relatively easy since its dominant effect on backscatter strength is essentially a near uniform change in amplitude at all lower incident angles.

5.2 m Estimation

For the estimation of m the input vector given to the network is the backscatter strength for incident angles ranging from 71 to 88 degrees in 1 degree increments, where each input to the network corresponds to a specific incident angle. Angles lower than 71 degrees are not required since the dominant effect of nm is at higher angles, and exclusion of the lower angles helps to reduce sensitivity of the estimation to changes in μ. Angles of 89 and 90 degrees gave backscatter strength values greater

Table 5.2: m network training

Learning Gain	Iterations	Average Percent Error
1	2000	4.55
0.9	2000	1.78
0.8	2000	1.45
0.7	2000	1.35
0.6	2000	1.19
0.5	2000	1.10
0.4	2000	1.03
0.3	2000	1.02
0.2	2000	0.98
0.1	2000	0.91
0.05	2000	0.88
0.0	1000	0.87

than 1 and were excluded. An angle increment of 1 degree was found to be sufficient to obtain small estimation errors. In the m network the parameter being estimated is m, but as mentioned previously a linear relationship is being used to obtain n given a value for m. The error could possibly be reduced by using a smaller increment, but this will also require a larger network. The data vector is normalized to reduce the effect of changes in μ by subtracting the backscatter strength at the angle of 71 degrees from all other points in the vector. Also, the data point corresponding to 71 degrees in the data vector is set to 1 to provide the network with an adjustable offset capability. As previously mentioned, the values of m may become too large for the tanh function, so the network is trained to estimate $m/4$. Thus the final estimate of m is given by the network output multiplied by 4.

The final configuration for the m network has 18 inputs to each of the nodes in the first layer and 8 nodes in each of the two hidden layers. The m estimation program is given in Appendix D.2. The final training sequence is shown in Table 5.2, and an average percent error of 0.87 was obtained with training off. The table indicates that the network trained rapidly to an error of about 5 percent, but then slowly decreased to its final value. This indicates that the error surface is shallow in the vicinity of the minimum, which will cause difficulty in training with noisy signals. Recall that all three parameters (μ, m, and δ) are being varied randomly, so this network has successfully differentiated between the effect of m and δ on the slope of the backscatter strength curve at high angles. The probability density function of the percent estimation error is shown in Figure 5.2, and indicates that for the majority of the test data the estimation error is below 2 percent.

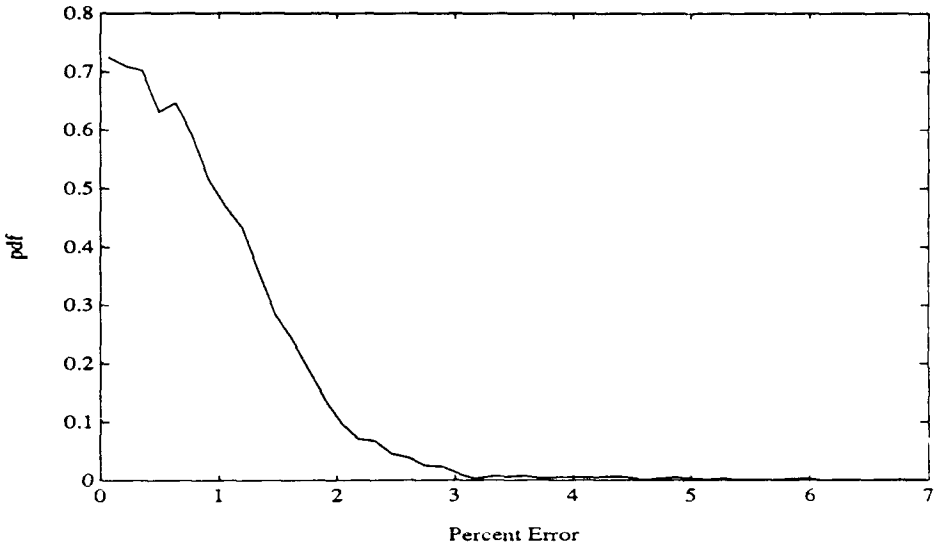

Figure 5.2: Probability density function of the percent estimation error for the m estimating program. The scattering strength data was obtained by randomly varying δ, μ, and m.

5.3 δ Estimation

For the estimation of δ the input vector given to the network is the backscatter strength for incident angles ranging from 71 to 88 degrees, and an increment of 0.5 degrees was required to obtain low estimation errors. As with the m parameter, angles lower than 71 degrees are not required since the dominant effect of δ is at higher angles, and exclusion of the lower angles helps to reduce sensitivity of the estimation to changes in μ. The data vector for the δ network is also normalized using the 71 degree data point, and this point is set to 1 to provide the network with an adjustable offset capability.

The final configuration for the δ network has 36 inputs to each of the nodes in the first layer, and 7 nodes in each of the two hidden layers. The δ estimation program is given in Appendix D.3. The final training sequence is shown in Table 5.3, and an average percent error of 5.16 was obtained with training off. Training of the δ network proved to be more difficult than for m and μ. The training sequence shows rapid convergence to an error of about 20 percent, but a much slower convergence to smaller errors indicating a very flat error surface near the minimum. Using a network with only 18 inputs to the first layer yielded a minimum error of about 15 percent without the adjustable offset capability (71 degree data point = 1), and about 12 percent with this capability. In earlier tests, when μ and m were held constant, an error of 1 percent was achieved, but this became as large as 20 percent when different values of μ and nm were used in testing than those used during training. Figure 5.3 shows the probability density function of the percent estimation error for the δ network, and indicates that for the majority of the test data the estimation error is below about 10 percent.

Table 5.3: δ network training

Learning Gain	Iterations	Average Percent Error
1.6	4000	17.1
1.5	4000	12.6
1.4	4000	8.74
1.3	4000	7.89
1.2	4000	7.70
1.1	4000	7.10
1.0	4000	6.98
0.9	4000	7.05
0.8	4000	6.72
0.7	4000	6.50
0.6	4000	6.07
0.5	4000	6.03
0.4	4000	6.09
0.3	4000	5.99
0.2	4000	5.80
0.1	4000	5.53
0.05	4000	5.51
0.0	1000	5.16

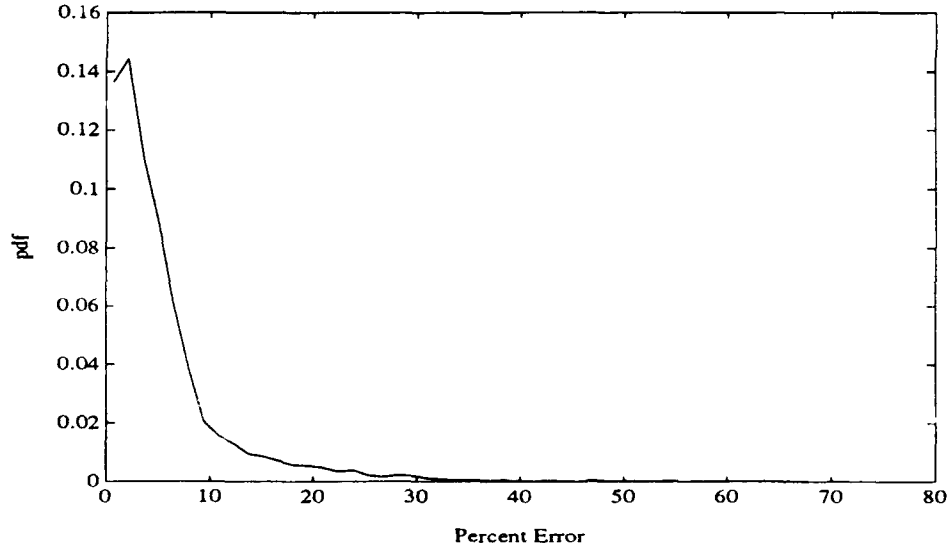

Figure 5.3: Probability density function of the percent estimation error for the δ estimating program. The scattering strength data was obtained by randomly varying δ, μ, and m.

Chapter 6

Summary

This work has demonstrated the ability to estimate certain parameters in the BISSM [1] acoustic scattering model using artificial backscatter data. The six parameters of interest are μ, n, m, σ, δ_x, and δ_y. It was desired to estimate these parameters given inputs of backscatter strength, m_{bs}, the incident angle, θ, and the azimuth angle, ϕ. These inputs potentially can be obtained from backscatter and bathymetric data collected by sidescan or multibeam acoustic imaging systems. Of the six parameters only three could be estimated: n and m are highly correlated, σ has minimal effect on the backscatter strength at 1.2 kHz, and δ_x and δ_y are highly correlated. δ_x and δ_y were assumed to be equal, giving a single parameter δ, and this eliminated ϕ from the monostatic version of the BISSM equation. A linear relationship was assumed between n and m, allowing m or n to be independently estimated. Feedforward neural networks were used to estimate the three parameters, μ, m, and δ, given backscatter strength values as a function of the incident angle. Using backscatter data generated with the BISSM equation, the networks successfully estimated μ and m with less than 1 percent average error, and δ with about 5 percent average error.

Further development of the neural networks used in this project will be required for application to real data. Real data will necessarily be corrupted with noise, and will seldom have ground truth information available. Without ground truth, real data cannot be used to train the networks. One approach to improving the parameter estimates in the presence of noise would be to analyze the noise character in real data. Then, synthetic noise of similar character can be used to contaminate artificially generated data, and the networks can be retrained to improve their performance with noisy data. Also, the number and range of incident angles available from real data depends on the bottom morphology as well as the survey system, with the result that the networks may not have a full input data vector. Further testing is required to determine the effect on the network's estimates in the event of missing input data points.

Bibliography

[1] J. Caruthers, R. Sandy, and J. Novarini, "Modified bistatic scattering stength model (BISSM2)," Tech. Rep. SP 023:200:90, Naval Oceanographic and Atmospheric Research Laboratory, Aug. 1990.

[2] J. Nelder and R. Mead *Computer Journal*, vol. 7, p. 308.

[3] R. Brent, *Algorithms for Minimization without Derivatives.* Englewood Cliffs, NJ: Prentice Hall, 1973.

[4] E. Polak, *Computational Methods in Optimization.* New York: Academic Press, 1971.

[5] D. Jacobs, *The State of the Art in Numerical Analysis.* London: Academic Press, 1977.

[6] A. Kolmogorov, "On the representation of continuous functions of many variables by superposition of continuous functions of one function and addition," *Dokl. Akad. Navk. USSR*, vol. 14, pp. 953-956, 1957.

[7] B. Widrow and M. A. Lehr, "30 years of adaptive neural networks: Perceptron, madaline, and backpropagation," *IEEE Proceedings*, pp. 1415–1442, Sept. 1990.

Appendix A

Neural Network Subroutine

```
/*  nn.c

    Brian Bourgeois    Created:  26AUG91    Last Mod: 26AUG91
This neural net sub is designed to take a fixed vector
length data as input and produce a single output variable,
representing some mapping from R^n to R^1.  Full connectivity
is used, and backpropagation is the learning
algorithm.  The ability to 'jar' the weights during learning
is included (annealing).  A call to nn causes a single pass
through the network.

All controlling information is passed through vectors arch
(architecture) and train.  Data is passed thru in, des, and est.
in is the input vector, des is the desired vector, and est is
the estimate vector.  The network weights are passed via taps.

In the vector arch, it is possible to specify the number
 on inputs to each of the first layer nodes, and the number of
 nodes in each layer.  A variable for specifying the number of
 layers is used, but the program is only set up for 3 layers
 at this time.

In the vector train, the learning (adaptive) gain is
 specified, which is typically less than 1.  The learning switch
 controls whether the node errors are computed and weights updated.
  The annealing gain controls the amount of noise added to the
 weights in a iteration.  train[3] and train[4] are not used in
 this sub, but in the controlling program.
*/


/* train[0] learning gain */
```

```
/* train[1] learning switch, 1 = on */
/* train[2] annealing gain, 0 = off */
/* train[3] 0 = init taps, 1 = load from taps.in */
/* train[4] Number of repetitions for this set */

/* arch[0] = Number of layers */
/* arch[1] = Number of neurons in first layer */
/* arch[2] = Number of neurons in second layer */
/* arch[3] = Number of neurons in output layer */
/* arch[4] = Number of inputs to first layer */

#include "nn.h"

/*************************************************/
/*************************************************/

int nn(in,des,est,arch,train,taps)
double *in; /* in = input vector */
double *des; /* des = desired vector */
double *est; /* est = estimated vector */
long *arch; /* network configuration parameters */
double *train; /* network training parameters */
double ***taps; /* network weights */
{

/****** declare variables *******/

    /* network configuration */
/* arch[0] = number of layers of neurons */
/* arch[1] = layer 1 length */
/* arch[2] = layer 2 length */
/* arch[3] = output layer length */
/* arch[4] = Number of inputs to first layer */
/* train[0] learning gain */
/* train[1] learning switch, 1 = on */
/* train[2] annealing gain, 0 = off */
/* train[3] 0 = init taps, 1 = load from taps.in */
/* train[4] Number of repetitions for this set */

long    no_layers,  /* Number of network layers with neurons */
        *no_neurons,/* Number of neurons in each layer */
        *no_inputs, /* Number of inputs for neuron in given layer*/
        layer,      /* Current layer number */
        node,       /* Current node number */
        tap,        /* Current tap number in a neuron */
```

```
        elem;       /* Node in next layer, for error calc */
double  sum,        /* Summer output for a neuron */
        **node_out, /* Outputs on each node in nn */
        **err,      /* Error for each neuron */
        corr;       /* Tap correction for updata */

    /* Misc variables */
long    ctr1, /* counter */
        ctr2, /* counter */
        ctr3; /* counter */
double  temp;

/****** Load up architecture variables ******/
no_layers = arch[0];

no_neurons = (long *)malloc(sizeof(long)*(int)arch[0]);
    if(no_neurons == NULL){
    printf("no_neurons allocation error {nn}\n");
    return;
    }
no_neurons[0] = arch[1];
no_neurons[1] = arch[2];
no_neurons[2] = arch[3];

no_inputs = (long *)malloc(sizeof(long)*(int)arch[0]);
    if(no_inputs == NULL){
    printf("no_inputs allocation error {nn}\n");
    return;
    }
no_inputs[0] = arch[4]; /* Inputs to first layer */
no_inputs[1] = no_neurons[0]; /* Inputs to second layer */
    /* same as no_neurons in first layer for full connectivity */
no_inputs[2] = no_neurons[1]; /* Inputs to output layer */
    /* same as no_neurons in second layer for full connectivity */

/****** Memory allocation ******/

node_out = (double **)malloc(sizeof(double **)*no_layers);
    if(node_out == NULL){
    printf("node_out allocation error, level 1\n");
    return;
    } /* if node_out == NULL */

for(ctr1=0;ctr1<no_layers;ctr1++){
node_out[ctr1] = (double *)malloc(sizeof(double)*no_neurons[ctr1]);
```

```
    if(node_out[ctr1] == NULL){
    printf("node_out allocation error, level 2\n");
    return;
    } /* if node_out[] == NULL */
} /* for ctr1 */

err = (double **)malloc(sizeof(double **)*no_layers);
    if(err == NULL){
    printf("err allocation error, level 1\n");
    return;
    } /* if err == NULL */

for(ctr1=0;ctr1<no_layers;ctr1++){
err[ctr1] = (double *)malloc(sizeof(double)*no_neurons[ctr1]);
    if(err[ctr1] == NULL){
    printf("err allocation error, level 2\n");
    return;
    } /* if err[] == NULL */
} /* for ctr1 */


/************ Functional Part of Program *************/

/***** Forward Sweep through network *****/

    /* Layer 1 */
layer = 0;
for(node=0;node<no_neurons[layer];node++){
    sum=0;
    for(tap=0;tap<no_inputs[layer];tap++){
        sum += taps[layer][node][tap]*in[tap];
    } /* for tap */
    node_out[layer][node] = tanh(sum);
} /* for node */

    /* Following Layers */
for(layer=1;layer<no_layers;layer++){
    for(node=0;node<no_neurons[layer];node++){
        sum=0;
for(tap=0;tap<no_inputs[layer];tap++){
            sum += taps[layer][node][tap]*node_out[layer-1][tap];
        } /* for tap */
        node_out[layer][node] = tanh(sum);
    } /* for node */
} /* for layer */
```

```
    /* Estimated Outputs */

for(ctr1=0;ctr1<arch[3];ctr1++){
    est[ctr1] = node_out[no_layers-1][ctr1];
}

/* printf("train[1] = %f\n",train[1]);  */
if(train[1] == 1){   /* Do this next section for learning on only */

/***** Backward Sweep through network *****/

/* Compute Errors */

    /* Output Layer */
layer = no_layers-1;
for(node=0;node<no_neurons[layer];node++){
err[layer][node] = (des[node]-est[node])*(1-est[node]*est[node]);
}

    /* Following Layers */
for(layer=no_layers-2;layer>-1;layer--){
    for(node=0;node<no_neurons[layer];node++){
        sum = 0;
        for(elem=0;elem<no_neurons[layer+1];elem++){
            sum += err[layer+1][elem]*taps[layer+1][elem][node];
        } /* for elem */
    temp = node_out[layer][node]*node_out[layer][node];
    err[layer][node] = sum*(1-temp);
    } /* for node */
} /* for layer */

/* Update filter weights */

    /* All but first layer */
for(layer=no_layers-1;layer>0;layer--){
    for(node=0;node<no_neurons[layer];node++){
for(tap=0;tap<no_inputs[layer];tap++){
        corr = 2.*train[0]*err[layer][node]*node_out[layer-1][tap];
        taps[layer][node][tap] += corr;
        } /* for tap */
    } /* for node */
} /* for layer */

    /* first layer */
```

```
layer = 0;
for(node=0;node<no_neurons[layer];node++){
    for(tap=0;tap<no_inputs[layer];tap++){
        corr = 2.*train[0]*err[layer][node]*in[tap];
        taps[layer][node][tap] += corr;
    } /* for tap */
} /* for node */

/* Jog taps: annealing method */
if(train[1]==1 && train[2]>0){
/* printf("Jog taps\n"); */
for(layer=0;layer<no_layers;layer++){
    for(node=0;node<no_neurons[layer];node++){
    for(tap=0;tap<no_inputs[layer];tap++){
    taps[layer][node][tap] += train[2]*(((double)rand()/RAND_MAX)-.5);
    } /* for tap */
    } /* for node */
} /* for layer */
} /* if train[1] && train[2] */

} /* if train[1] == 1, Learning on */


/******* End Sub ******/

/* free memory */

for(ctr1=0;ctr1<no_layers;ctr1++){
    free(node_out[ctr1]);
}
free(node_out);

for(ctr1=0;ctr1<no_layers;ctr1++){
    free(err[ctr1]);
}
free(err);

free(no_neurons);
free(no_inputs);

return;
}
```

Appendix B

Neural Network Training Program

```
/* del.c

    Brian Bourgeois    Created:  13SEP91    Last Mod: 23SEP91
This is the driving main program for the neural net sub 'nn.c'.
The taps array is created and maintained in this program, and
network architecture and training motif are also specified here
and passed to the network sub.  Each call to nn does a single
pass through the network.

This program also provides an area for calling a data generation
routine, for specifying the input and desired signals.  */

#include "nn.h"
#include "bism.h"

extern int nn(double*, double*, double *, long *,
 double *, double ***);
extern int bism(double *ss, double *theta, double *model);
/* exclude this for cc */

main(int argc, char *argv[])
/* main(argc,argv)
int argc;
char *argv[]; */
{

/******* Declare variables ********/

    /* Data vectors */
double *in;      /* network input vector */
double *des;     /* network desired vector */
```

```
double *est;    /* network estimated vector */

    /* network configuration */
long *arch;  /* network configuration parameters */
double *train; /* network training parameters */
double ***taps;   /* network weights */
long no_layers; /* Number of network layers with neurons */
long *no_neurons; /* Number of neurons in each layer */
long *no_inputs; /* Number of inputs for a neuron in a given layer */

    /* Training variables */
long num_sets; /* Number of training sets in train.dat file */
long set_cnt; /* Current training set */
double error; /* Output error */
double mse; /* Output mse */
double perr; /* desired signal squared */
double errdev; /* error std deviation */
double *iterr; /* Error for each iteration */

    /* Misc variables */

long    ctr1, /* counter */
        ctr2,  * counter */
        ctr3, /* counter */
        flag,
        temp;
FILE    *ftaps_in, /* Input taps file, with arch header */
        *ftaps_out, /* Output taps file, with arch header */
        *ftrain, /* File with training instructions */
        *fdefarch, /* default network architecture file */
    *fdefmodel, /* Default model parameters file */
*fiterr; /* iteration error data dump file */

/* Model variables */

double *theta, /* incident angle array */
*ss, /* scattering strength data */
model[9]; /* model parameters */

/* Model Parameters
model[0] = size of data vectors theta and ss
model[1] = n,      Ratio of sound speeds
model[2] = m,      Ratio of densities
model[3] = mu,     Lambert coefficient
model[4] = sigma,  Microscale heights roughness
```

```
model[5] = phi,     azimuth angle in radians
model[6] = delx,    RMS slope deviation along track, radians
model[7] = dely,    RMS slope deviation across track, radians
model[8] = k;       Wavenumber
*/

long    cnt1,    /* counter */
        cnt2,    /* counter */
        min,     /* minimum thet angle */
        max;     /* maximum thet angle */
 double step;    /* thet angle step size */

/* Assign model parameters */

flag = 0;
min = 71; /* minimum theta in degrees */
max = 88; /* max theta : 60 to 89 for del test */
step = .5; /* theta angle step size */
/* compute vector length */
model[0] = (double)(max - min + 1)/step ; /* size of theta vector */
/* Note: No of Inputs to network should be same as model[0] */

/****** Obtain network configuration data ******/

    /* Allocate storage for controlling parameters */

arch = (long *)malloc(sizeof(long)*(int)5);
    if(arch == NULL){
    printf("arch allocation error\n");
    return;
    }

train = (double *)malloc(sizeof(double)*(int)5);
    if(train == NULL){
    printf("train allocation error\n");
    return;
    }

    /* Pick up first training set.  This is done first to determine
 (from init field) if the architecture will be specified by the file
 defarch.dat, or if weights from a previous execution will be used,
 and thus the arch will be contained in a file taps.in */

ftrain = fopen("train.dat","r");
if(ftrain == NULL){
```

```
    printf("Training file does not exist -- terminating\n");
    return;
    }

/* read in no of training sets */
fscanf(ftrain," %ld",&num_sets);

/* read in first training set */
for(ctr1=0;ctr1<5;ctr1++){
    fscanf(ftrain," %lf",&train[ctr1]);
    } /* for ctr1 */

/* train[0] learning gain */
/* train[1] learning switch, 1 = on */
/* train[2] annealing gain, 0 = off */
/* train[3] 0 = init taps, 1 = load from taps.in */
/* train[4] Number of repetitions for this set */

if(train[3] == 1){
/* Read in architecture from existing taps file, taps.in */
ftaps_in = fopen("taps.in","r");
if(ftaps_in == NULL){
    printf("taps.in does not exist -- terminating\n");
    return;
    }
for(ctr1=0;ctr1<5;ctr1++){
    fscanf(ftaps_in," %ld",&arch[ctr1]);
    } /* for ctr1 */
} /* if train[3] */
else{
    /* use default network architecture vector */
fdefarch = fopen("defarch.dat","r");
if(fdefarch == NULL){
    printf("defarch.dat does not exist -- terminating\n");
    return;
    }
for(ctr1=0;ctr1<5;ctr1++){
    fscanf(fdefarch," %ld",&arch[ctr1]);
    } /* for ctr1 */
fclose(fdefarch);

/* arch[0] = Number of layers */
/* arch[1] = Number of neurons in first layer */
/* arch[2] = Number of neurons in second layer */
/* arch[3] = Number of neurons in output layer */
```

```
/* arch[4] = Number of inputs to first layer */
} /* else */

    /* Load up architecture variables */

no_layers = arch[0];

no_neurons = (long *)malloc(sizeof(long)*(int)arch[0]);
    if(no_neurons == NULL){
    printf("no_neurons allocation error\n");
    return;
    }
no_neurons[0] = arch[1];
no_neurons[1] = arch[2];
no_neurons[2] = arch[3];

no_inputs = (long *)malloc(sizeof(long)*(int)arch[0]);
    if(no_inputs == NULL){
    printf("no_inputs allocation error\n");
    return;
    }
no_inputs[0] = arch[4]; /* Inputs to first layer  */
/* Note this should be same as length of ss vector */
no_inputs[1] = no_neurons[0]; /* Inputs to second layer */
    /* same as no_neurons in first layer for full connectivity */
no_inputs[2] = no_neurons[1]; /* Inputs to output layer */
    /* same as no_neurons in second layer for full connectivity */

/****** Allocate storage ******/

in = (double *)malloc(sizeof(double)*(int)arch[4]);
    if(in == NULL){
    printf("in allocation error\n");
    return;
    }

des = (double *)malloc(sizeof(double)*(int)arch[3]);
    if(des == NULL){
    printf("des allocation error\n");
    return;
    }

est = (double *)malloc(sizeof(double)*(int)arch[3]);
    if(est == NULL){
    printf("est allocation error\n");
```

```
    return;
    }

theta = (double *)malloc(sizeof(double)*(int)model[0]);
if(theta == NULL){
    printf("theta allocation error\n");
    return;
    }

ss = (double *)malloc(sizeof(double)*(int)model[0]);
if(ss == NULL){
    printf("ss allocation error\n");
    return;
    }

    /* nn weights allocation is a bit more involved */
taps = (double ***)malloc(sizeof(double ***)*no_layers);
    if(taps == NULL){
    printf("taps allocation error, level 1\n");
    return;
    } /* if taps == NULL */

for(ctr1=0;ctr1<no_layers;ctr1++){
taps[ctr1] = (double **)malloc(sizeof(double **)*no_neurons[ctr1]);
    if(taps[ctr1] == NULL){
    printf("taps allocation error, level 2\n");
    return;
    } /* if taps[] == NULL */
} /* for ctr1 */

for(ctr1=0;ctr1<no_layers;ctr1++){
for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
taps[ctr1][ctr2] = (double *)malloc(sizeof(double)*no_inputs[ctr1]);
    if(taps[ctr1][ctr2] == NULL){
    printf("taps allocation error, level 3\n");
    return;
    } /* if taps[][] == NULL */
} /* for ctr2 */
} /* for ctr1 */

/*****************************************/
/* Working Area    MAIN */

    /****** Load taps array ******/
if(train[3]==1){
```

```
/* Load taps from taps.in and close file */
for(ctr1=0;ctr1<no_layers;ctr1++){
    for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
for(ctr3=0;ctr3<no_inputs[ctr1];ctr3++){
fscanf(ftaps_in," %le",&taps[ctr1][ctr2][ctr3]);
        } /* for tap */
    } /* for node */
} /* for layer */
fclose(ftaps_in);
} /* if train[3] */
else{
/* Initialize taps with small random numbers */
for(ctr1=0;ctr1<no_layers;ctr1++){
for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
for(ctr3=0;ctr3<no_inputs[ctr1];ctr3++){
taps[ctr1][ctr2][ctr3] = .5*(((double)rand()/RAND_MAX)-.5);
 /* -.5 to .5 */
} } }
/* Print weights
for(ctr1=0;ctr1<no_layers;ctr1++){
for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
for(ctr3=0;ctr3<no_inputs[ctr1];ctr3++){
printf("%f\n",taps[ctr1][ctr2][ctr3]);
} } } */
} /* else */

/* Assign incident angles */
for(cnt1=0;cnt1<model[0];cnt1++){
theta[cnt1] = (double)(min + cnt1*step) * PI / 180.;
/* printf("theta = %f\n",theta[cnt1]); */
}

/* Load nominal model parameters */

fdefmodel = fopen("defmodel.dat","r");
if(fdefmodel == NULL){
    printf("defmodel.dat does not exist -- terminating\n");
    return;
    }
for(ctr1=1;ctr1<9;ctr1++){
    fscanf(fdefmodel," %lf",&model[ctr1]);
    } /* for ctr1 */
fclose(fdefmodel);

/* model[1] = n, nom .99 */
```

```
/* model[2] = m, nom 1.4 */
/* model[3] = mu, nom .002 */
/* model[4] = sigma, nom .01 */
/* model[5] = phi, no effect for delx=dely */
/* model[6] = delx, nom .05236 radians */
/* model[7] = dely, nom .05236 radians */
/* model[8] = k, nom 5.0265 for 1200 Hz @ 1500 m/s */


    /******* MAIN PROGRAM LOOP *******/

/* Print net arch to stdio */
printf("Net Architecture:\n");
printf("No. layers    = %ld\n",arch[0]);
printf("Layer 1 nodes = %ld\n",arch[1]);
printf("Layer 2 nodes = %ld\n",arch[2]);
printf("Layer 3 nodes = %ld\n",arch[3]);
printf("No. inputs    = %ld\n\n",arch[4]);

/* Print theta range to stdio */
printf("Theta parameters:\n");
printf("min  = %ld\n", min);
printf("max  = %ld\n", max);
printf("step = %lf\n\n", step);

/* Print default model parameters */
printf("Default model parameters:\n");
printf("n     = %lf\n",model[1]);
printf("m     = %lf\n",model[2]);
printf("mu    = %lf\n",model[3]);
printf("sigma = %lf\n",model[4]);
printf("del   = %lf\n\n",model[6]);


/* Print the number of training sets to stdio */
printf("Number of sets = %ld\n",num_sets);

for(set_cnt=0;set_cnt<num_sets;set_cnt++){
/* This is loop for number of training sets */

/* Get new training set, after first pass only */
if(set_cnt>0){
    for(ctr1=0;ctr1<5;ctr1++){
flag = fscanf(ftrain," %lf",&train[ctr1]);
    } /* for ctr1 */
```

```
if(flag != 1){
printf("EOF reached in train.dat: terminating\n");
return;
} /* if flag */
} /* if set_cnt */

/* Print training setup for this training set */
printf("Set No. %ld\n",set_cnt);
printf("Learn Gain   = %lf\n",train[0]);
printf("Learn Switch = %lf\n",train[1]);
printf("Anneal Gain  = %lf\n",train[2]);
printf("Init Switch  = %lf\n",train[3]);
printf("No. reps     = %lf\n",train[4]);

/* Assign iteration error memory */

iterr = (double *)malloc(sizeof(double)*(int)train[4]);
    if(iterr == NULL){
    printf("iterr allocation error\n");
    return;
    }

/* Initialize error */
perr = 0;

for(temp=0;temp<(long)train[4];temp++){
/* Loop for repetition of same training set */

/* Obtain input and estimated data */

/* Compute a random value for delx&dely, scale in the range
 of .01745 to .08727 for the desired signal (1 to 5 degrees) */

do{
model[6] = (((double)rand()/RAND_MAX)); /* range is 0 to 1 */
model[6] = .08727 * model[6];
}while(model[6] < .01745);

model[7] = model[6];
des[0] = model[6];

/* Randomize the nm parameters to try and make del estimation
 independent of them */

do{
```

```
model[2] = (((double)rand()/RAND_MAX)); /* range is 0 to 1 */
model[2] = .8 + model[2];
}while(model[2] < 1.2); /* range is 1.2 to 1.8 */

model[1] =  model[2] * .3833 + .51; /* range is .97 to 1.2 */

/* Randomize the mu parameter: range .0002 t0 .02 */

do{
model[3] = (((double)rand()/RAND_MAX)); /* range is 0 to 1 */
model[3] = .02 * model[3];
}while(model[3] < .0002);

/* Call bism.c to generate input vector, scale for 0 to 1 */
/* theta is from 71 to 88 degress for del test, 89 degrees
gives too large a value of ss (2.5) for the program to handle.
Using 71 degrees vice 60 to reduce sensitivty to changes in mu.
Larger errors were obtained at 60 deg even when noramlizing the
curve to the start angle magnitude, and this is appraently due
to the curvature caused by mu */

flag = bism(ss, theta, model);
if(flag == 1){
printf("Error in bism sub\n");
return;
}

for(ctr1=0;ctr1<model[0];ctr1++){
in[ctr1]=ss[ctr1]-ss[0];  /* normalize ampl for 71 deg */
}
in[0]=1; /* Give a constant to play with */

/* Call to network subroutine    */

nn(in,des,est,arch,train,taps);
/* printf("interset %ld, est = %le\n",temp,est[0]); */


/* Computer errors */
iterr[temp] = fabs(des[0]-est[0])*100./des[0];
perr += iterr[temp];

} /* for temp */

/* Compute err std deviation */
```

```
perr = perr / (double)train[4];
errdev = 0;
for(ctr1=0;ctr1<train[4];ctr1++){
errdev += (iterr[ctr1] - perr) * (iterr[ctr1] - perr);
}
errdev = sqrt(errdev / ((double)train[4]-1));

/* Print results of network for end of current set */

printf("avg perr = %lf\n",perr);
printf("perr dev = %lf\n",errdev);
printf("last des = %lf\n",des[0]);
printf("last est = %lf\n",est[0]);
printf("last n   = %lf\n",model[1]);
printf("last m   = %lf\n",model[2]);
printf("last mu  = %lf\n",model[3]);

/* Print errors to ascii file */
fiterr = fopen("iterr.out","w");
if(fiterr == NULL){
printf("Cannot open iterr file {w}\n");
} /* if fiterr == NULL */
else{
for(ctr1=0;ctr1<train[4];ctr1++){
fprintf(fiterr,"%lf\n",iterr[ctr1]);
} /* for ctr1 */
} /* else ftaps_out */
fclose(fiterr);

free(iterr);

} /* for set_cnt */

/******** Finish up *********/

/* Close training file */

fclose(ftrain);

/* Save arch and taps to output file */
ftaps_out = fopen("taps.out","w");
if(ftaps_out == NULL){
printf("Cannot open taps file {w}\n");
} /* if ftaps_out == NULL */
else{
```

```
printf("Dumping arch and taps\n\n\n");
for(ctr1=0;ctr1<5;ctr1++){
fprintf(ftaps_out," %ld",arch[ctr1]);
}
for(ctr1=0;ctr1<no_layers;ctr1++){
    for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
for(ctr3=0;ctr3<no_inputs[ctr1];ctr3++){
fprintf(ftaps_out," %le",taps[ctr1][ctr2][ctr3]);
        } /* for tap */
    } /* for node */
} /* for layer */
} /* else ftaps_out */
fclose(ftaps_out);

/* Free memory */
free(in);
free(des);
free(est);
free(arch);
free(train);
free(theta);
free(ss);

    /* free tap memory */
for(ctr1=0;ctr1<no_layers;ctr1++){
    for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
    free(taps[ctr1][ctr2]);
    }
}
for(ctr1=0;ctr1<no_layers;ctr1++){
    free(taps[ctr1]);
}
free(taps);

return;
}
```

Appendix C

BISSM Subroutine

```
/* Brian Bourgeois, Written 26AUG91, Modified 26AUG91 */
/* Program bism.c */
/* This program will compute the various parts, and the total
    scattering strength based upon the bissm2 model and its various
    parameters.  The output is a vector of scattering values for
    a specified range of theta (incident ray angle).   */

#include "bism.h"

int bism(ss, theta, model)
double  *ss,
*theta,
*model;
{

/* Model Parameters
model[0] = size of data vectors theta and ss
model[1] = n,       Ratio of sound speeds
model[2] = m,       Ratio of densities
model[3] = mu,      Lambert coefficient
model[4] = sigma,   Microscale heights roughness
model[5] = phi,     azimuth angle in radians
model[6] = delx,    RMS slope deviation along track, radians
model[7] = dely,    RMS slope deviation across track, radians
model[8] = k;       Wavenumber
*/

/* Model intermediate variables */
double  temp1,  /* interm variable */
        temp2,  /* interm variable */
        temp3,  /* interm variable */
        R,      /* Rayleigh coefficient */
```

```
        R_a,    /* part of Rayleight computation */
        R_b,    /* part of Rayleight computation */
        g,      /* g */
        ae,     /* angular dependent exponential part */
        ref,    /* reflective component */
        lam;    /* Lambert component */

/* Misc variables */
long    cnt1,   /* counter */
        cnt2,   /* counter */
flag;  /* error flag */

/* Note: The smallest value of thet must be large enough with
respect to n for the Rayleigh reflection coefficient calculation.
This can be made program adjustable in the future, but for now lets
just flag it */

if((model[1] < 1) && (theta[0] < acos(model[1]))){
    printf("Error in theta range\n");
    flag = 1;
    return flag;
    }

/* Scattering Stength Computations */

for(cnt1=0;cnt1<model[0];cnt1++){

    /* Rayleigh Coefficient */
    R_a = model[2] * sin(theta[cnt1]);
    temp1 = model[1] * model[1];
    temp2 = cos(theta[cnt1]);
    temp2 = temp2 * temp2;
    R_b = sqrt(temp1 - temp2);
    R = (R_a - R_b)/(R_a + R_b);

    /* compute g */
    temp1 = 2 * model[4] * model[8] * sin(theta[cnt1]);
    g = temp1 * temp1;

    /* Compute angular dependent exp part */
    temp1 = cos(model[5]);
    temp1 = (temp1 * temp1)/(model[6]*model[6]);
    temp2 = sin(model[5]);
    temp2 = (temp2 * temp2)/(model[7]*model[7]);
    temp1 = -.5 * (temp1 + temp2);
```

```
        temp2 = cos(theta[cnt1])/sin(theta[cnt1]);
        temp1 = temp1 * (temp2 * temp2);
        temp1 = exp(temp1)/(8 * PI * model[6] * model[7]);
        temp2 = sin(theta[cnt1]);
        temp2 = (temp2 * temp2) * (temp2 * temp2);
        ae = temp1/temp2;

        /* Compute reflective component */

        ref = (R * R) * exp(-1*g) * ae;

        /* Compute lambert component */

        temp1 = sin(theta[cnt1]);
        lam = model[3] * (temp1 * temp1);

        /* Compute total scattering strength */
        ss[cnt1] = ref + lam;

        } /* for cnt1, scatter stength loop */

    return;
    }
```

Appendix D

Parameter Estimation Programs

D.1 μ Parameter Estimation

```
/* muest.c

    Brian Bourgeois    Created:  25SEP91    Last Mod: 25SEP91

This program is used to estimate the mu parameter of the
BISSM algorithm given a vector that is scattering strength
vs. angle of incidence. The program loads its network
architecture and taps from file taps.in.  Note that the
number of ss points provided to this program, and their
respective angles is fixed by the network architecture and
its training.  An area is provided in this program to call
for data input.  */

#define RAND_MAX 2147483647 /* 2  31 -1 */
#define PI  3.14159265358979

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>

extern int nn(double*, double*, double *, long *,
 double *, double ***);
extern int bism(double *ss, double *theta, double *model);
/* exclude this for cc */

main(int argc, char *argv[])
/* main(argc,argv)
int argc;
```

```
char *argv[]; */
{

/******* Declare variables ********/

    /* network configuration */
long arch[5];  /* network configuration parameters */
double train[5]; /* network parameters */
double ***taps;   /* network weights */
long no_layers; /* Number of network layers with neurons */
long *no_neurons; /* Number of neurons in each layer */
long *no_inputs; /* Number of inputs for a neuron in a given layer */

    /* data variables */
double des;    /* network desired signal */
double est;    /* network estimated signal */
double perr; /* desired signal squared */
double errdev; /* error std deviation */
double *iterr; /* Error for each iteration */

    /* Misc variables */

long    ctr1, /* counter */
        ctr2, /* counter */
        ctr3, /* counter */
        flag,
        temp;
double  ftemp;
FILE    *ftaps_in; /* Input taps file, with arch header */
long    cnt1,   /* counter */
        min,    /* minimum thet angle */
        max;    /* maximum thet angle */
 double step;   /* thet angle step size */

/* Model variables */

double *theta, /* incident angle array */
*ss, /* scattering strength data */
model[9]; /* model parameters */

/* Model Parameters
model[0] = size of data vectors theta and ss
model[1] = n,       Ratio of sound speeds
model[2] = m,       Ratio of densities
model[3] = mu,      Lambert coefficient
```

```
model[4] = sigma,  Microscale heights roughness
model[5] = phi,    azimuth angle in radians
model[6] = delx,   RMS slope deviation along track, radians
model[7] = dely,   RMS slope deviation across track, radians
model[8] = k;      Wavenumber
*/

/* Initialize variables */
flag = 0;
train[0]=0; /* Not used */
train[1]=0; /* Not used */
train[2]=0; /* Not used */
train[3]=0; /* Not used */
train[4]=1000; /* No. of iterations to do */

/* Compute theta vector length */
/* This must match the training motif used for the network in use */

min = 15; /* minimum theta in degrees */
max = 60; /* max theta */
step = 2; /* theta angle step size */
/* compute vector length */
model[0] = (double)(max - min + 1)/step ; /* size of theta vector */

/****** Obtain network configuration data ******/

/* Read in architecture from taps file, taps.in */
ftaps_in = fopen("taps.in","r");
if(ftaps_in == NULL){
    printf("taps.in does not exist -- terminating\n");
    return;
    }
for(ctr1=0;ctr1<5;ctr1++){
    fscanf(ftaps_in," %ld",&arch[ctr1]);
    } /* for ctr1 */

/* arch[0] = Number of layers */
/* arch[1] = Number of neurons in first layer */
/* arch[2] = Number of neurons in second layer */
/* arch[3] = Number of neurons in output layer */
/* arch[4] = Number of inputs to first layer */

    /* Load up architecture variables */

no_layers = arch[0];
```

```
no_neurons = (long *)malloc(sizeof(long)*(int)arch[0]);
    if(no_neurons == NULL){
    printf("no_neurons allocation error\n");
    return;
    }
no_neurons[0] = arch[1];
no_neurons[1] = arch[2];
no_neurons[2] = arch[3];

no_inputs = (long *)malloc(sizeof(long)*(int)arch[0]);
    if(no_inputs == NULL){
    printf("no_inputs allocation error\n");
    return;
    }
no_inputs[0] = arch[4]; /* Inputs to first layer  */
/* Note this should be same as length of ss vector */
no_inputs[1] = no_neurons[0]; /* Inputs to second layer */
    /* same as no_neurons in first layer for full connectivity */
no_inputs[2] = no_neurons[1]; /* Inputs to output layer */
    /* same as no_neurons in second layer for full connectivity */

/****** Allocate storage ******/

theta = (double *)malloc(sizeof(double)*(int)model[0]);
if(theta == NULL){
    printf("theta allocation error\n");
    return;
    }

ss = (double *)malloc(sizeof(double)*(int)model[0]);
if(ss == NULL){
    printf("ss allocation error\n");
    return;
    }

iterr = (double *)malloc(sizeof(double)*(int)train[4]);
    if(iterr == NULL){
    printf("iterr allocation error\n");
    return;
    }

    /* nn weights allocation is a bit more involved */
taps = (double ***)malloc(sizeof(double ***)*no_layers);
    if(taps == NULL){
```

```
    printf("taps allocation error, level 1\n");
    return;
    } /* if taps == NULL */

for(ctr1=0;ctr1<no_layers;ctr1++){
taps[ctr1] = (double **)malloc(sizeof(double **)*no_neurons[ctr1]);
    if(taps[ctr1] == NULL){
    printf("taps allocation error, level 2\n");
    return;
    } /* if taps[] == NULL */
} /* for ctr1 */

for(ctr1=0;ctr1<no_layers;ctr1++){
for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
taps[ctr1][ctr2] = (double *)malloc(sizeof(double)*no_inputs[ctr1]);
    if(taps[ctr1][ctr2] == NULL){
    printf("taps allocation error, level 3\n");
    return;
    } /* if taps[][] == NULL */
} /* for ctr2 */
} /* for ctr1 */

/************************************************/
/* Working Area of MAIN */

    /****** Load taps array ******/
/* Load taps from taps.in and close file */
for(ctr1=0;ctr1<no_layers;ctr1++){
    for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
for(ctr3=0;ctr3<no_inputs[ctr1];ctr3++){
fscanf(ftaps_in," %le",&taps[ctr1][ctr2][ctr3]);
        } /* for tap */
    } /* for node */
} /* for layer */
fclose(ftaps_in);

/* Assign incident angles */
for(cnt1=0;cnt1<model[0];cnt1++){
theta[cnt1] = (double)(min + cnt1*step) * PI / 180.;
/* printf("theta = %f\n",theta[cnt1]); */
}

/* Load nominal model parameters */

model[1] = .99;
```

```
model[2] = 1.4;
model[3] = .002;
model[4] = .01;
model[5] = 0;
model[6] = .05236;
model[7] = .05236;
model[8] = 5.0265;

    /******* MAIN PROGRAM LOOP *******/

/* Loop for multiple estimations  */

printf("WORKING\n");
for(temp=0;temp<(long)train[4];temp++){

/* Obtain input and estimated data */

/* Compute a random value for delx&dely scale in the range
 of .01745 to .08727 for the desired signal (1 to 5 degrees) */

do{
model[6] = (((double)rand()/RAND_MAX)); /* range is 0 to 1 */
model[6] = .08727 * model[6];
}while(model[6] < .01745);

model[7] = model[6];

/* Compute a random value for nm.  Note that n and m are
 computed as being linearly related */

do{
model[2] = (((double)rand()/RAND_MAX)); /* range is 0 to 1 */
model[2] = .8 + model[2];
}while(model[2] < 1.2); /* range is 1.2 to 1.8 */

model[1] =  model[2] * .3833 + .51; /* range is .97 to 1.2 */

/* Randomize the mu parameter: range .0002 t0 .02 */

do{
model[3] = (((double)rand()/RAND_MAX)); /* range is 0 to 1 */
model[3] = .02 * model[3];
}while(model[3] < .0002);

des = model[3];
```

```
/* Call bism.c to generate input vector, scale for range 0 to 1 */

flag = bism(ss, theta, model);
if(flag == 1){
printf("Error in bism sub\n");
return;
}

/* Adjust ss data for proper network operation, as per training */
/* For 10 to 60 deg, ss ranges from 3*10^-6 to 3*10^-2 */
/* Scale up from .03 to .3 */
for(ctr1=0;ctr1<model[0];ctr1++){
ss[ctr1]=ss[ctr1]*10. - .15;
}

/* Call to network subroutine    */

nn(ss,&des,&est,arch,train,taps);

/* Compute errors */
iterr[temp] = fabs(des-est)*100./des;

} /* for temp */

/* Compute err and its std deviation */
perr = 0;
for(ctr1=0;ctr1<train[4];ctr1++){
perr += iterr[ctr1];
}
perr = perr / (double)train[4];
errdev = 0;
for(ctr1=0;ctr1<train[4];ctr1++){
errdev += (iterr[ctr1] - perr) * (iterr[ctr1] - perr);
}
errdev = sqrt(errdev / ((double)train[4]-1));

/* Print results of network test */

printf("avg perr = %lf\n",perr);
printf("perr dev = %lf\n",errdev);
printf("last des = %lf\n",des);
printf("last est = %lf\n",est);

/******** Finish up *********/
```

```
/* Free memory */
free(iterr);
free(theta);
free(ss);
free(no_neurons);
free(no_inputs);

    /* free tap memory */
for(ctr1=0;ctr1<no_layers;ctr1++){
    for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
    free(taps[ctr1][ctr2]);
    }
}
for(ctr1=0;ctr1<no_layers;ctr1++){
    free(taps[ctr1]);
}
free(taps);

return;
}
```

D.2 *nm* Parameter Estimation

```
/* nmest.c

    Brian Bourgeois    Created:  25SEP91    Last Mod: 25SEP91

This program is used to estimate the nm parameter of the BISSM
algorithm given a vector that is scattering strength vs. angle
of incidence.  The program loads its network architecture and
taps from file taps.in.  Note that the number of ss points
provided to this program, and their respective angles is fixed
by the network architecture and its training.
An area is provided in this program to call for data input.
*/

#define RAND_MAX 2147483647 /* 2  31 -1 */
#define PI  3.14159265358979

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <ctype.h>
```

```
#include <stdlib.h>

extern int nn(double*, double*, double *, long *,
 double *, double ***);
extern int bism(double *ss, double *theta, double *model);
/* exclude this for cc */

main(int argc, char *argv[])
/* main(argc,argv)
int argc;
char *argv[]; */
{

/******* Declare variables ********/

    /* network configuration */
long arch[5];  /* network configuration parameters */
double train[5]; /* network parameters */
double ***taps;   /* network weights */
long no_layers; /* Number of network layers with neurons */
long *no_neurons; /* Number of neurons in each layer */
long *no_inputs; /* Number of inputs for a neuron in a given layer */

    /* data variables */
double des;    /* network desired signal */
double est;    /* network estimated signal */
double perr; /* desired signal squared */
double errdev; /* error std deviation */
double *iterr; /* Error for each iteration */

    /* Misc variables */

long    ctr1, /* counter */
        ctr2, /* counter */
        ctr3, /* counter */
        flag,
        temp;
double  ftemp;
FILE    *ftaps_in; /* Input taps file, with arch header */
long    cnt1,   /* counter */
        min,    /* minimum thet angle */
        max;    /* maximum thet angle */
 double step;   /* thet angle step size */

/* Model variables */
```

```
double *theta, /* incident angle array */
*ss, /* scattering strength data */
model[9]; /* model parameters */

/* Model Parameters
model[0] = size of data vectors theta and ss
model[1] = n,       Ratio of sound speeds
model[2] = m,       Ratio of densities
model[3] = mu,      Lambert coefficient
model[4] = sigma,   Microscale heights roughness
model[5] = phi,     azimuth angle in radians
model[6] = delx,    RMS slope deviation along track, radians
model[7] = dely,    RMS slope deviation across track, radians
model[8] = k;       Wavenumber
*/

/* Initialize variables */
flag = 0;
train[0]=0; /* Not used */
train[1]=0; /* Not used */
train[2]=0; /* Not used */
train[3]=0; /* Not used */
train[4]=1000; /* No. of iterations to do */

/* Compute theta vector length */
/* This must match the training motif used for the network in use */

min = 71; /* minimum theta in degrees */
max = 88; /* max theta */
step = 1; /* theta angle step size */
/* compute vector length */
model[0] = (double)(max - min + 1)/step ; /* size of theta vector */

/****** Obtain network configuration data ******/

/* Read in architecture from taps file, taps.in */
ftaps_in = fopen("taps.in","r");
if(ftaps_in == NULL){
    printf("taps.in does not exist -- terminating\n");
    return;
    }
for(ctr1=0;ctr1<5;ctr1++){
    fscanf(ftaps_in," %ld",&arch[ctr1]);
    } /* for ctr1 */
```

```
/* arch[0] = Number of layers */
/* arch[1] = Number of neurons in first layer */
/* arch[2] = Number of neurons in second layer */
/* arch[3] = Number of neurons in output layer */
/* arch[4] = Number of inputs to first layer */

    /* Load up architecture variables */

no_layers = arch[0];

no_neurons = (long *)malloc(sizeof(long)*(int)arch[0]);
    if(no_neurons == NULL){
    printf("no_neurons allocation error\n");
    return;
    }
no_neurons[0] = arch[1];
no_neurons[1] = arch[2];
no_neurons[2] = arch[3];

no_inputs = (long *)malloc(sizeof(long)*(int)arch[0]);
    if(no_inputs == NULL){
    printf("no_inputs allocation error\n");
    return;
    }
no_inputs[0] = arch[4]; /* Inputs to first layer  */
/* Note this should be same as length of ss vector */
no_inputs[1] = no_neurons[0]; /* Inputs to second layer */
    /* same as no_neurons in first layer for full connectivity */
no_inputs[2] = no_neurons[1]; /* Inputs to output layer */
    /* same as no_neurons in second layer for full connectivity */

/****** Allocate storage ******/

theta = (double *)malloc(sizeof(double)*(int)model[0]);
if(theta == NULL){
    printf("theta allocation error\n");
    return;
    }

ss = (double *)malloc(sizeof(double)*(int)model[0]);
if(ss == NULL){
    printf("ss allocation error\n");
    return;
    }
```

```
iterr = (double *)malloc(sizeof(double)*(int)train[4]);
    if(iterr == NULL){
    printf("iterr allocation error\n");
    return;
    }

    /* nn weights allocation is a bit more involved */
taps = (double ***)malloc(sizeof(double ***)*no_layers);
    if(taps == NULL){
    printf("taps allocation error, level 1\n");
    return;
    } /* if taps == NULL */

for(ctr1=0;ctr1<no_layers;ctr1++){
taps[ctr1] = (double **)malloc(sizeof(double **)*no_neurons[ctr1]);
    if(taps[ctr1] == NULL){
    printf("taps allocation error, level 2\n");
    return;
    } /* if taps[] == NULL */
} /* for ctr1 */

for(ctr1=0;ctr1<no_layers;ctr1++){
for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
taps[ctr1][ctr2] = (double *)malloc(sizeof(double)*no_inputs[ctr1]);
    if(taps[ctr1][ctr2] == NULL){
    printf("taps allocation error, level 3\n");
    return;
    } /* if taps[][] == NULL */
} /* for ctr2 */
} /* for ctr1 */

/************************************************/
/* Working Area of MAIN */

    /****** Load taps array ******/
/* Load taps from taps.in and close file */
for(ctr1=0;ctr1<no_layers;ctr1++){
    for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
for(ctr3=0;ctr3<no_inputs[ctr1];ctr3++){
fscanf(ftaps_in," %le",&taps[ctr1][ctr2][ctr3]);
        } /* for tap */
    } /* for node */
} /* for layer */
fclose(ftaps_in);
```

```
/* Assign incident angles */
for(cnt1=0;cnt1<model[0];cnt1++){
theta[cnt1] = (double)(min + cnt1*step) * PI / 180.;
/* printf("theta = %f\n",theta[cnt1]); */
}

/* Load nominal model parameters */

model[1] = .99;
model[2] = 1.4;
model[3] = .002;
model[4] = .01;
model[5] = 0;
model[6] = .05236;
model[7] = .05236;
model[8] = 5.0265;

    /******* MAIN PROGRAM LOOP *******/

/* Loop for multiple estimations  */

printf("WORKING\n");
for(temp=0;temp<(long)train[4];temp++){

/* Obtain input and estimated data */

/* Randomize delx&dely, scale in the range of .01745 to .08727
 (1 to 5 degrees) */

do{
model[6] = (((double)rand()/RAND_MAX)); /* range is 0 to 1 */
model[6] = .08727 * model[6];
}while(model[6] < .01745);

model[7] = model[6];

/* Compute a random value for nm.  Note that n and m are
 computed as being linearly related */

do{
model[2] = (((double)rand()/RAND_MAX)); /* range is 0 to 1 */
model[2] = .8 + model[2];
}while(model[2] < 1.2); /* range is 1.2 to 1.8 */
```

```
model[1] =  model[2] * .3833 + .51; /* range is .97 to 1.2 */
des = model[2]/4.;

/* Randomize the mu parameter: range .0002 t0 .02 */

do{
model[3] = (((double)rand()/RAND_MAX)); /* range is 0 to 1 */
model[3] = .02 * model[3];
}while(model[3] < .0002);

/* Call bism.c to generate input vector, scale for range 0 to 1 */

flag = bism(ss, theta, model);
if(flag == 1){
printf("Error in bism sub\n");
return;
}

/* normalize ampl for starting incident angle to reduce
 mu effect on estimate */
ftemp = ss[0];
for(ctr1=0;ctr1<model[0];ctr1++){
ss[ctr1]=ss[ctr1]-ftemp;
}
ss[0]=1; /* Constant input node for network */

/* Call to network subroutine    */

nn(ss,&des,&est,arch,train,taps);

/* Compute errors */
iterr[temp] = fabs(des-est)*100./des;

} /* for temp */

/* Compute err and its std deviation */
perr = 0;
for(ctr1=0;ctr1<train[4];ctr1++){
perr += iterr[ctr1];
}
perr = perr / (double)train[4];
errdev = 0;
for(ctr1=0;ctr1<train[4];ctr1++){
errdev += (iterr[ctr1] - perr) * (iterr[ctr1] - perr);
}
```

```
errdev = sqrt(errdev / ((double)train[4]-1));

/* Print results of network test */

printf("avg perr = %lf\n",perr);
printf("perr dev = %lf\n",errdev);
printf("last des = %lf\n",4.*des);
printf("last est = %lf\n",4.*est);

/******** Finish up *********/

/* Free memory */
free(iterr);
free(theta);
free(ss);
free(no_neurons);
free(no_inputs);

    /* free tap memory */
for(ctr1=0;ctr1<no_layers;ctr1++){
    for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
    free(taps[ctr1][ctr2]);
    }
}
for(ctr1=0;ctr1<no_layers;ctr1++){
    free(taps[ctr1]);
}
free(taps);

return;
}
```

D.3 δ Parameter Estimation

```
/* delest.c

    Brian Bourgeois    Created:  23SEP91    Last Mod: 24SEP91

This program is used to estimate the del parameter of the
BISSM algorithm given a vector that is scattering strength
vs. angle of incidence.  Due to the dependence of the delx
and dely parameters, only a single parameter, del, is used
wherein delx = dely = del.  The program loads its network
architecture and taps from file taps.in.  Note that the
```

```
number of ss points provided to this program, and their
respective angles is fixed by the network architecture
and its training.  An area is provided in this program to
call for data input.  */

#define RAND_MAX 2147483647 /* 2  31 -1 */
#define PI  3.14159265358979

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>

extern int nn(double*, double*, double *, long *,
 double *, double ***);
extern int bism(double *ss, double *theta, double *model);
/* exclude this for cc */

main(int argc, char *argv[])
/* main(argc,argv)
int argc;
char *argv[]; */
{

/******* Declare variables ********/

    /* network configuration */
long arch[5];  /* network configuration parameters */
double train[5]; /* network parameters */
double ***taps;   /* network weights */
long no_layers; /* Number of network layers with neurons */
long *no_neurons; /* Number of neurons in each layer */
long *no_inputs; /* Number of inputs for a neuron in a given layer */

    /* data variables */
double des;    /* network desired signal */
double est;    /* network estimated signal */
double perr; /* desired signal squared */
double errdev; /* error std deviation */
double *iterr; /* Error for each iteration */

    /* Misc variables */

long    ctr1, /* counter */
```

```
        ctr2, /* counter */
        ctr3, /* counter */
        flag,
        temp;
double  ftemp;
FILE    *ftaps_in; /* Input taps file, with arch header */
long    cnt1,   /* counter */
        min,    /* minimum thet angle */
        max;    /* maximum thet angle */
 double step;   /* thet angle step size */

/* Model variables */

double *theta, /* incident angle array */
*ss, /* scattering strength data */
model[9]; /* model parameters */

/* Model Parameters
model[0] = size of data vectors theta and ss
model[1] = n,       Ratio of sound speeds
model[2] = m,       Ratio of densities
model[3] = mu,      Lambert coefficient
model[4] = sigma,   Microscale heights roughness
model[5] = phi,     azimuth angle in radians
model[6] = delx,    RMS slope deviation along track, radians
model[7] = dely,    RMS slope deviation across track, radians
model[8] = k;       Wavenumber
*/

/* Initialize variables */
flag = 0;
train[0]=0; /* Not used */
train[1]=0; /* Not used */
train[2]=0; /* Not used */
train[3]=0; /* Not used */
train[4]=1000; /* No. of iterations to do */

/* Compute theta vector length */
/* This must match the training motif used for the network in use */

min = 71; /* minimum theta in degrees */
max = 88; /* max theta */
step = .5; /* theta angle step size */
/* compute vector length */
model[0] = (double)(max - min + 1)/step ; /* size of theta vector */
```

```
/****** Obtain network configuration data ******/

/* Read in architecture from taps file, taps.in */
ftaps_in = fopen("taps.in","r");
if(ftaps_in == NULL){
    printf("taps.in does not exist -- terminating\n");
    return;
    }
for(ctr1=0;ctr1<5;ctr1++){
    fscanf(ftaps_in," %ld",&arch[ctr1]);
    } /* for ctr1 */

/* arch[0] = Number of layers */
/* arch[1] = Number of neurons in first layer */
/* arch[2] = Number of neurons in second layer */
/* arch[3] = Number of neurons in output layer */
/* arch[4] = Number of inputs to first layer */

    /* Load up architecture variables */

no_layers = arch[0];

no_neurons = (long *)malloc(sizeof(long)*(int)arch[0]);
    if(no_neurons == NULL){
    printf("no_neurons allocation error\n");
    return;
    }
no_neurons[0] = arch[1];
no_neurons[1] = arch[2];
no_neurons[2] = arch[3];

no_inputs = (long *)malloc(sizeof(long)*(int)arch[0]);
    if(no_inputs == NULL){
    printf("no_inputs allocation error\n");
    return;
    }
no_inputs[0] = arch[4]; /* Inputs to first layer  */
/* Note this should be same as length of ss vector */
no_inputs[1] = no_neurons[0]; /* Inputs to second layer */
    /* same as no_neurons in first layer for full connectivity */
no_inputs[2] = no_neurons[1]; /* Inputs to output layer */
    /* same as no_neurons in second layer for full connectivity */

/****** Allocate storage ******/
```

```
theta = (double *)malloc(sizeof(double)*(int)model[0]);
if(theta == NULL){
    printf("theta allocation error\n");
    return;
    }

ss = (double *)malloc(sizeof(double)*(int)model[0]);
if(ss == NULL){
    printf("ss allocation error\n");
    return;
    }

iterr = (double *)malloc(sizeof(double)*(int)train[4]);
    if(iterr == NULL){
    printf("iterr allocation error\n");
    return;
    }

    /* nn weights allocation is a bit more involved */
taps = (double ***)malloc(sizeof(double ***)*no_layers);
    if(taps == NULL){
    printf("taps allocation error, level 1\n");
    return;
    } /* if taps == NULL */

for(ctr1=0;ctr1<no_layers;ctr1++){
taps[ctr1] = (double **)malloc(sizeof(double **)*no_neurons[ctr1]);
    if(taps[ctr1] == NULL){
    printf("taps allocation error, level 2\n");
    return;
    } /* if taps[] == NULL */
} /* for ctr1 */

for(ctr1=0;ctr1<no_layers;ctr1++){
for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
taps[ctr1][ctr2] = (double *)malloc(sizeof(double)*no_inputs[ctr1]);
    if(taps[ctr1][ctr2] == NULL){
    printf("taps allocation error, level 3\n");
    return;
    } /* if taps[][] == NULL */
} /* for ctr2 */
} /* for ctr1 */

/*********************************************/
```

```
/* Working Area of MAIN */

    /****** Load taps array ******/
/* Load taps from taps.in and close file */
for(ctr1=0 ctr1<no_layers;ctr1++){
    for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
for(ctr3=0;ctr3<no_inputs[ctr1];ctr3++){
fscanf(ftaps_in," %le",&taps[ctr1][ctr2][ctr3]);
        } /* for tap */
    } /* for node */
} /* for layer */
fclose(ftaps_in);

/* Assign incident angles */
for(cnt1=0;cnt1<model[0];cnt1++){
theta[cnt1] = (double)(min + cnt1*step) * PI / 180.;
/* printf("theta = %f\n",theta[cnt1]); */
}

/* Load nominal model parameters */

model[1] = .99;
model[2] = 1.4;
model[3] = .002;
model[4] = .01;
model[5] = 0;
model[6] = .05236;
model[7] = .05236;
model[8] = 5.0265;

    /******* MAIN PROGRAM LOOP *******/

/* Loop for multiple estimations  */

printf("WORKING\n");
for(temp=0;temp<(long)train[4];temp++){

/* Obtain input and estimated data */

/* Compute a random value for delx&dely, scale in the range of
 .01745 to .0872 for the desired signal (1 to 5 degrees) */

do{
model[6] = (((double)rand()/RAND_MAX)); /* range is 0 to 1 */
model[6] = .08727 * model[6];
```

```
}while(model[6] < .01745);

model[7] = model[6];
des = model[6];

/* Compute a random value for nm.  Note that n and m are
 computed as being linearly related */

do{
model[2] = (((double)rand()/RAND_MAX)); /* range is 0 to 1 */
model[2] = .8 + model[2];
}while(model[2] < 1.2); /* range is 1.2 to 1.8 */

model[1] =  model[2] * .3833 + .51; /* range is .97 to 1.2 */

/* Randomize the mu parameter: range .0002 t0 .02 */

do{
model[3] = (((double)rand()/RAND_MAX)); /* range is 0 to 1 */
model[3] = .02 * model[3];
}while(model[3] < .0002);

/* Call bism.c to generate input vector, scale for range 0 to 1 */

flag = bism(ss, theta, model);
if(flag == 1){
printf("Error in bism sub\n");
return;
}

/* normalize ampl for starting incident angle to reduce
 mu effect on estimate */
ftemp = ss[0];
for(ctr1=0;ctr1<model[0];ctr1++){
ss[ctr1]=ss[ctr1]-ftemp;
}
ss[0]=1; /* Constant input node for network */

/* Call to network subroutine    */

nn(ss,&des,&est,arch,train,taps);

/* Compute errors */
iterr[temp] = fabs(des-est)*100./des;
```

```
} /* for temp */


/* Compute err and its std deviation */
perr = 0;
for(ctr1=0;ctr1<train[4];ctr1++){
perr += iterr[ctr1];
}
perr = perr / (double)train[4];
errdev = 0;
for(ctr1=0;ctr1<train[4];ctr1++){
errdev += (iterr[ctr1] - perr) * (iterr[ctr1] - perr);
}
errdev = sqrt(errdev / ((double)train[4]-1));

/* Print results of network test */

printf("avg perr = %lf\n",perr);
printf("perr dev = %lf\n",errdev);
printf("last des = %lf\n",des);
printf("last est = %lf\n",est);

/******** Finish up *********/

/* Free memory */
free(iterr);
free(theta);
free(ss);
free(no_neurons);
free(no_inputs);

    /* free tap memory */
for(ctr1=0;ctr1<no_layers;ctr1++){
    for(ctr2=0;ctr2<no_neurons[ctr1];ctr2++){
    free(taps[ctr1][ctr2]);
    }
}
for(ctr1=0;ctr1<no_layers;ctr1++){
    free(taps[ctr1]);
}
free(taps);

return;
}
```

Distribution List

Commander, Naval Oceanography Command
Stennis Space Center, MS 39529-5004

Naval Oceanographic & Atmospheric Research Laboratory
Stennis Space Center MS 39529-5004
Attn:
Code 125L (10)
Code 125P
Code 240
Code 250
Code 300
Code 350
Code 351

REPORT DOCUMENTATION PAGE

Form Approved
OBM No. 0704-0188

1. Agency Use Only *(Leave blank).*	2. Report Date.	3. Report Type and Dates Covered.
	December 1991	Final

4. Title and Subtitle.

Neural Network Parameter Estimation for the Modified Bistatic Scattering Strength Model (BISSM)

5. Funding Numbers.

Program Element No. O&M,N
Project No.
Task No.
Accession No. DN251150
Work Unit No. 92001B

6. Author(s).

B.S. Bourgeois

7. Performing Organization Name(s) and Address(es).

Naval Oceanographic and Atmospheric Research Laboratory
Ocean Science Directorate
Stennis Space Center, Mississippi 39529-5004

8. Performing Organization Report Number.

NOARL Technical Note 214

9. Sponsoring/Monitoring Agency Name(s) and Address(es).

Commander, Naval Oceanography Command
Stennis Space Center, Mississippi 39529-5000

10. Sponsoring/Monitoring Agency Report Number.

NOARL Technical Note 214

11. Supplementary Notes.

12a. Distribution/Availability Statement.

Approved for public release; distribution is unlimited.

12b. Distribution Code.

13. Abstract *(Maximum 200 words).*

This technical note investigates the estimation of environmental parameters in the Bistatic Scattering Strength Model (BISSM) given backscatter strength and bathymetric data. A monostatic version of the model is derived, since this will be the form of data provided by acoustic imaging sensors. Feedforward neural networks, using the backpropagation learning algorithm, are used to perform the estimation of parameters for the nonlinear BISSM equation. The parameters that can be estimated are identified, and neural networks have been developed to estimate these parameters. Using noise-free artificial data generated with the BISSM equation, the networks provided excellent estimates of the desired parameters.

14. Subject Terms.

Bottom Scattering, Reverberation

15. Number of Pages.

71

16. Price Code.

17. Security Classification of Report.	18. Security Classification of This Page.	19. Security Classification of Abstract.	20. Limitation of Abstract.
Unclassified	Unclassified	Unclassified	SAR